ANNUAL REPORT

VALIDATION OF MULTIPROCESSOR SYSTEMS

Daniel P. Siewiorek
Zary Segall
Gary York
Thomas Kong
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

December 23, 1982

Table of Contents

Chapter 1
Voter Queue Length Experiments

1.1. Introduction

1.1.1. Background

In N-modular-redundancy (NMR) computer systems, the redundant modules are often computer-memory pairs. The computers communicate information to be voted on either by hardware voters [26], or by software voters running on the processors [8] [21]. Software voting has a number of distinct advantages over hardware voting, one of which is the flexibility of the voter. A software voter routine can change its expectations as the system changes, thereby improving the system reliability. Features such as dynamic reconfiguration have been shown to improve system reliability [32] [14]. Most of the research on NMR redundancy has made the assumption that the modules are synchronized [4]. This assumption does not hold for a large class of systems, and often it is very difficult to force processors to be tightly synchronized. Some people are beginning to realize that asynchronous systems offer distinct advantages in reliability [21], and simplicity. The problem remains though of how to design an asynchronous system that meets the reliability objectives.

1.1.2. Objectives

In an asynchronous NMR computer system, the processors will each have their own clock, and will make little or no effort to synchronize the clocks with each other. The random variation in clock speed, and the difference in process execution patterns will cause differences in the arrival times of the data to be voted on by the voters. The voters should be able to receive data asynchronously, so

that they can vote on the data when a majority of the processes have sent it. The voters must be able to store message values, so that one processor can be calculating the 10^{th} step in a procedure, while another processor can be working on the 12^{th} step. Eventually both processors should finish the procedure, but as long as no data dependencies exist one processor should not be forced to wait for another to finish a calculation. Even when data dependencies do exist, when a majority of the processor agree on the value of a step, there is no reason to wait for the rest of the processors to finish before continuing with the next step. In fact, waiting can reduce reliability if a processor is faulty, since it may never respond to the voter. There should, however, be a limit to the amount a processor should be allowed to get behind before it is considered faulty. The random variation may cause problems if one processor becomes hopelessly behind due to the variation. Experiments have been performed to discover the nature of how variations in process execution speed affect the amount a process gets behind the others. The effects of variation in process execution speed, as well as variation of the number of instructions executed between votes have been examined.

Three experiments have been performed. Each is designed to explore a different area of the synchronization problem. Experiment 1 has one process execute more instructions for every step in the experiment. This process is continuously slower. This experiment shows that the voter overhead increases as the slow process falls behind. Experiment 2 has one process slower for a period, followed by being faster for a period. These oscillations in process execution speed are realistic for some systems. Experiment 3 has one process slower for a period, followed by a period of normal speed. This experiment is most realistic, since processes are likely to fall behind in a system, but are not likely to speed up.

1.2. Experiment Description

A task to be performed is broken into equal subtasks. Each subtask is executed in order, with data being passed from one subtask to the next. It is assumed that each subtask has the exact same execution speed, and that only one word of data is passed from one part to the next.

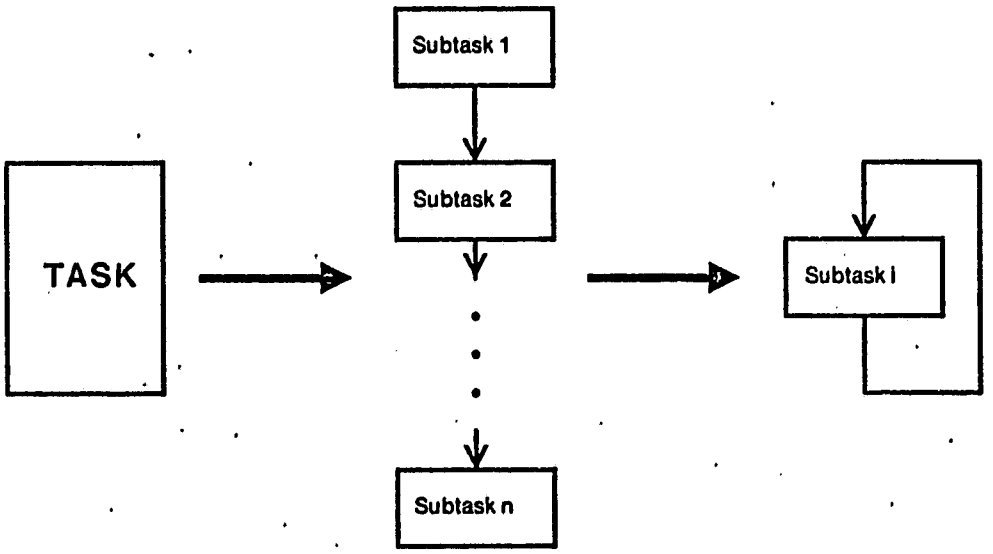

Figure 1-1: Experiment Task Partitioning

Since the subtasks all have the same execution speed, the task can be simulated by a loop that executes n times with a synthetic workload that takes subtask$_i$ time inside the loop. Figure 1-1 shows the partitioning. Each subtask is triplicated, and a vote occurs on the data passed between subtasks, yielding the structure in Figure 1-2.

1.3. Subtask Description

The triplicated subtasks all do the same function. They will calculate the i^{th} data value, send a copy of the data to each voter, and receive the voted value of the data from the associated voter. The

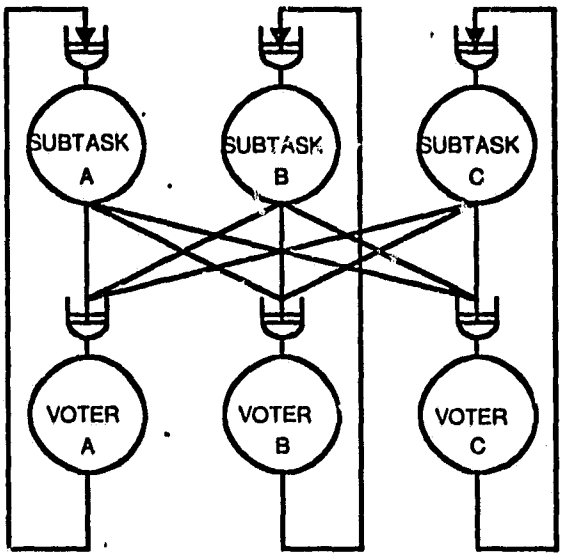

Figure 1-2: TMR Queue Length Experiment Structure

new data value is then used in calculating the $(i+1)^{st}$ data value.[1] The time each subtask takes to calculate the i^{th} data value is an experimental variable. Each of the triplicated subtasks could have a different calculation time for iteration i, allowing the simulation of variation in process execution speed. By varying the subtask speeds independently, synchronization issues can be explored.

1.4. Voter Description

The voter is also triplicated. Each voter accepts three words of data, one from each subtask; and votes on the data from the i^{th} iteration as soon as a majority (two for triplication) of the data values for iteration i are received by the voter. There are three input buffers (see Figure 1-3), one for each subtask, in which a voter can receive the data.The voter constantly looks for data from each subtask by examining the input buffers. A round-robin scheme is used to allow each subtask to have a "fair" opportunity to have its new data received by the voter.

---

[1] One can imagine wanting to pass more than one data value from one subtask to the next. This can be done with a more complicated voter. The entire state of a processor (or selected parts) could be passed as data, allowing a faulty processor to recover from a transient by accepting the voted state as its new state.

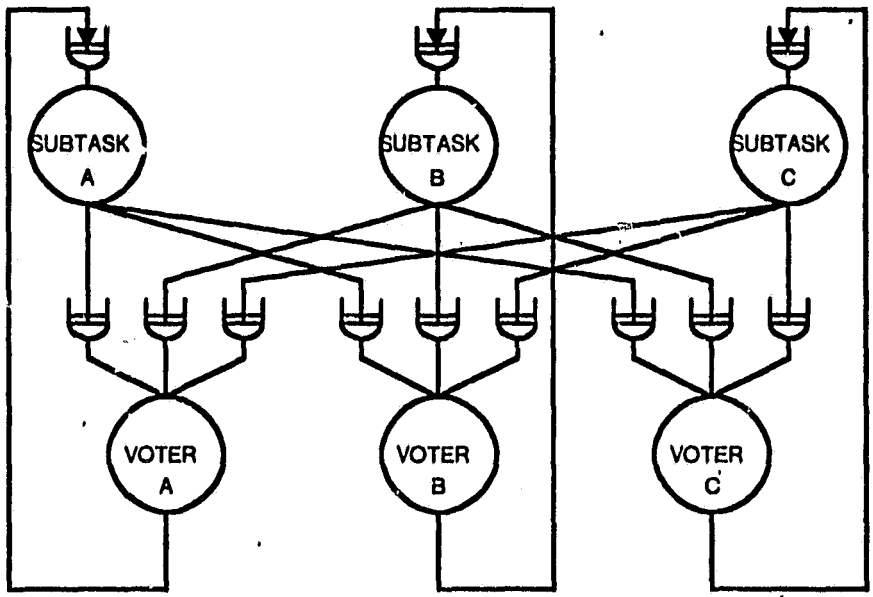

Figure 1-3: Voter Structure with Three Separate Input Buffers

As long as the subtasks have similar execution speeds, the voter should receive the i^{th} data value at approximately the same time for each subtask. However, if one subtask is slower than the other two, then the voter may receive the $(i+1)^{st}$ data value from a fast subtask before the slow subtask sends the i^{th} data value. (Remember that the voter will vote and send the voted data value as soon as a majority -two- of the values agree.) Since the voter now has data from two different iterations, it must be able to distinguish which data is associated with which iteration, and from which subtask. A voter queue is used to maintain this database. Each row in the queue contains information about:

1. which iteration this row represents.

2. whether data has arrived from each subtask.

3. what the data value is from a source subtask (if it has arrived).

The column the data is stored in implicitly identifies the associated destination subtask.

Each subtask must send the voter not only the data value, but also the iteration number. The voter

can then search for an iteration number in the voter queue to find the row where the data for this subtask belongs. If the iteration number is not found in the queue, then a row for this iteration is placed in the queue, and the data is placed in the row. When all of the data values for a particular row have arrived, the voter reports any errors found while voting and then removes the row from the queue.

The voter queue has a finite maximum length. If one subtask has not sent any data to the voter in the same period in which the other two subtasks have sent many data messages, then the voter queue could conceivably become full. The voter handles a full queue by removing the oldest row (associated with an iteration for which all the data has not arrived) from the queue, and then adding a row associated with the new iteration number. Errors are reported on the row removed from the queue. The maximum length of the queue can be large, so that the queue will never become full in experiments.

Since each subtask sends the iteration number, the messages need not arrive at the voter in order. On Cm* [12], the messages will always arrive in order, but some networks do not guarantee ordered arrival. The iteration number also is useful in identifying missing messages. A voter could miss a message entirely and still be able to associate later messages with the proper iteration. A reliability model of the voter, though, must take into account the probability of the iteration number being faulty.

1.5. Experiment One

The first experiment performed with the above subtask-voter paradigm was designed to measure the ability of the voter to synchronize the subtasks, when one subtask is continuously slower than the other subtasks. The frequency of voting (or granularity of the subtasks) was varied, and the execution speed of one subtask was varied. The queue lengths of the voters were recorded, as a measure of how far the slow subtask fell behind the two faster subtasks. The granularity of a subtask is defined as

how many operations must be performed to calculate a data value for a subtask iteration. The slower subtask performed 10% to 150% more operations in calculating the next value. The slower subtask represents a process that requires more execution time due to an instruction retry, or due to an interrupt that it must handle. In these situations, one subtask will be temporarily slower; but as these experiments show, it would be ill-advised to design a system where one subtask was continuously slower (this experiment shows design contraints for systems that have one continuously slower subtask). Each voter recorded the length of the voter queue every time a new iteration was received. The queue length information was sent as a message to a process that stored the data in a file. The recording of the queue length added some overhead to the voter, but each voter paid the same overhead cost.

The queue length is plotted versus the iteration number for various granularities, and various subtask degradation. The graphs are shown in Figures 1-4 to 1-7. For small granularity, one subtask can be up to 50% slower, and the queue length stays at one. This implies that the voter overhead is great enough so that the differences in speed is masked. For larger differences in speed, the queue length grows to a value, and then levels off. The queue length is bounded due to an increase in voter execution time as the queue length increases. The voter must search for the iteration number in the queue, and the search proceeds linearly. The subtask that is slower, will not pay this overhead cost since it has n messages waiting for processing, where n is the queue length.

As the granularity increases, the queue length grows more rapidly. Granularity is defined as the number of operations the normal subtasks must perform to prepare a data value. The slower subtask will perform 10% to 150% more operations in data value preparation. With granularity equal to 1024 (Figure 1-5), the 10% to 40% additional operations curves appear to be bounded, but the 50% additional operations curve is not bounded. The curves for granularity equal to 4K (Figure 1-6) and granularity equal to 16K (Figure 1-7) do not appear to have a bounded queue length. This in due to the fact that the voter overhead takes a smaller percentage of the total execution time for the larger granularity cases. The voter overhead is a fixed value for a specific queue length. When the slower

subtask takes approximately the same amount of time as the voter, then the voter overhead is significant in comparison to the subtask execution time. While the normal subtasks are waiting on the voter to generate a voted data value, the slower subtask can be calculating a data value for one of the old messages (when the queue length is greater than one, the subtask will have data values to calculate for all the messages in the queue).

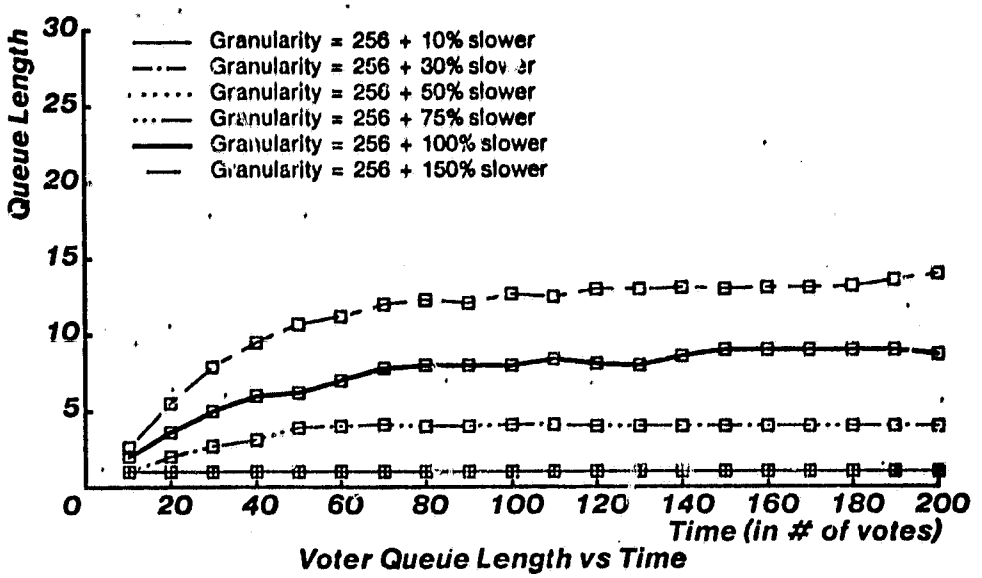

Figure 1-4: Granularity equal to 256, one subtask always slower

1.6. Experiment Two

The second experiment is a variation on the first experiment and was designed to explore the synchronizing nature of voters more fully. In this experiment, one subtask is slower than the other two subtasks by a percentage for a period of time, then the same subtask is faster than the other subtasks for the same period. The period was chosen to be 20 iterations. For example, subtask A will perform 10% more operations in calculating the first 20 data values, followed by performing 10% fewer operations for the next 20 iterations.

While the subtask is operating slower, the queue length should behave exactly the same as in

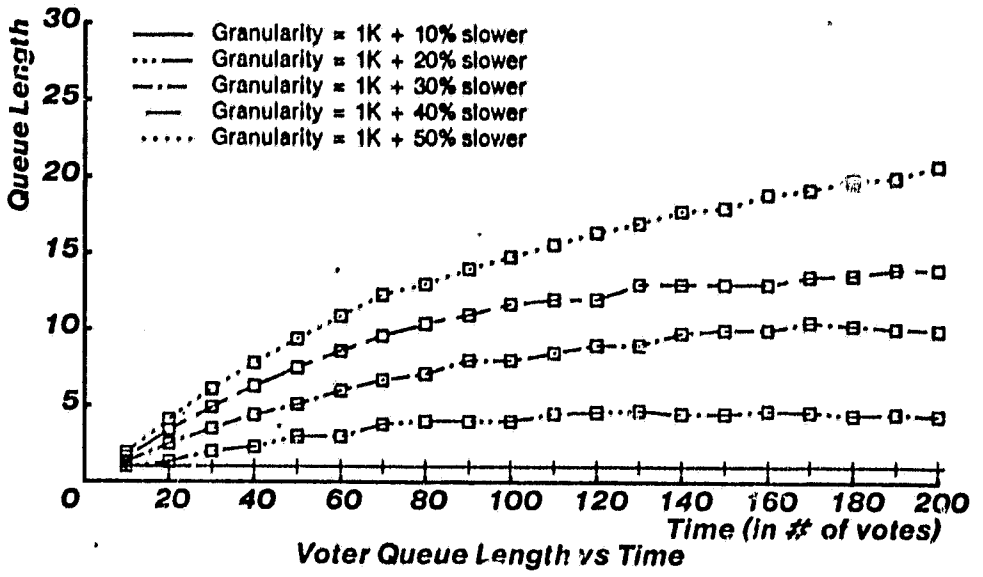

Figure 1-5: Granularity equal to 1024, one subtask always slower

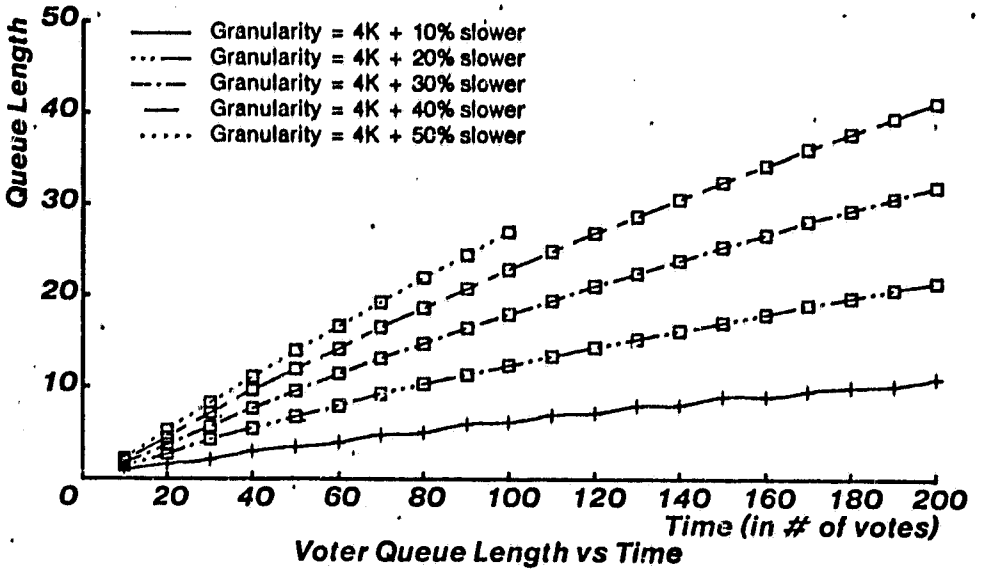

Figure 1-6: Granularity equal to 4K, one subtask always slower

experiment one. Once the subtask is faster than the others, then this subtask should quickly catch up, resulting in a decline in the queue length. The rate of decline in queue length should be greater than

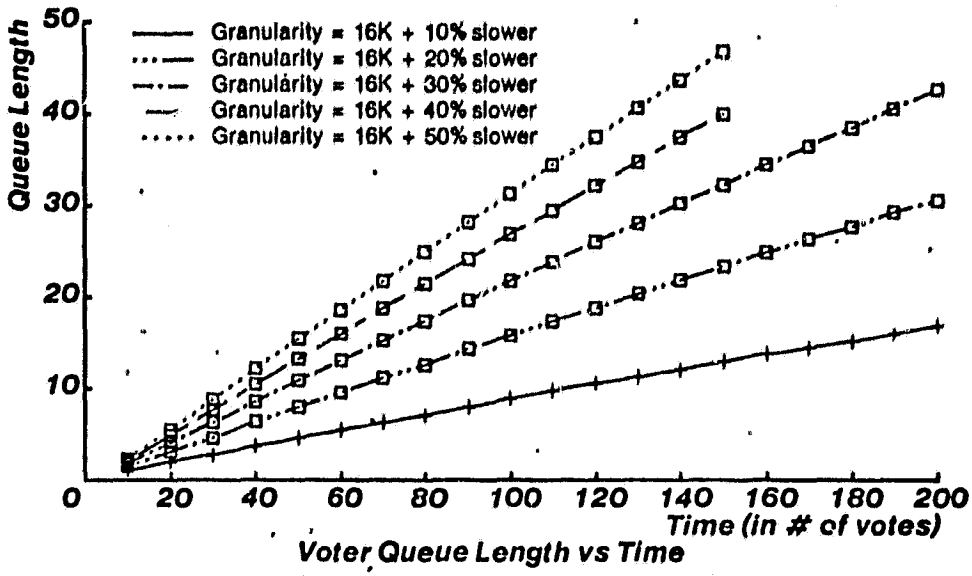

Figure 1-7: Granularity equal to 16K, one subtask always slower

the rate of increase, since when the queue has length greater than one, the subtask being varied does not have to wait for the voter to finish before beginning the next data value calculation.

This experiment is somewhat realistic, since random variation in processor speed is expected in any non-synchronous computer network. Non-uniform variations in process execution rate, such as local error correction procedures, may cause a temporary variation in subtask execution speed. The variation in these experiments, 10% to 100%, is realistic for a random variation in execution rate due to software error recovery, and the data yields some interesting insights into the nature of voting synchronization.

The first plot of queue length versus iteration number with granularity equal to 256, (Figure 1-8) shows the expected result. The queue length increases when subtask A is slower, and the rate of increase is the same as that form experiment one. As soon as subtask A begins executing fewer operations per iteration, the queue length declines rapidly, reaching queue length equal to one. When granularity equals 1024 (Figure 1-9), the same result is evident. The rate of queue length

decrease is greater than the rate of queue length increase. Figure 1-10 shows similar results. However, when subtask A is 50% faster, the queue length does go back to one, but the subtask barely gets to one before it begins to execute the greater number of operations. The rate of decline of queue length is greater than the rate of rise, but the rise continues for a much longer time.

If subtask A is executing 10% more operations for 20 iterations, then the following calculations should hold:

- time spent executing 10% more operations = 20*(110% of operation speed)
- time spent executing 10% fewer operations = 20*(90% of operation speed)
- percentage of time spent executing more operations = 55%
- percentage of time spent executing fewer operations = 45%

The subtask is spending 10% more time executing the long calculations. When the overhead of the voter is approximately equal to the additional time spent executing the longer calculations, then the queue length will become one just in time to begin executing the longer calculation iterations. It appears as if this balance is met when granularity equal to 4K, and the subtask executes 50% more operations followed by 50% fewer operations. If the granularity is increased to 16K (Figure 1-11) then the queue length is not restored to one, and there is a net increase in the queue length over time. Upon careful evaluation, this result is expected, however it is disheartening to see a net increase in queue length when intuition would indicate a bounded queue length.

1.7. Experiment Three

The third experiment is similar to experiment two, except it represents a more realistic class of synchronization problems.. A subtask that is performing a calculation, may experience a temporary slowdown, followed by a period of normal behavior such as a subtask which has to perform a recovery routine because of a bus error, or has to perform a one time operating system task. Is the processor running the subtask doomed to stay behind, or will it eventually catch up even though it

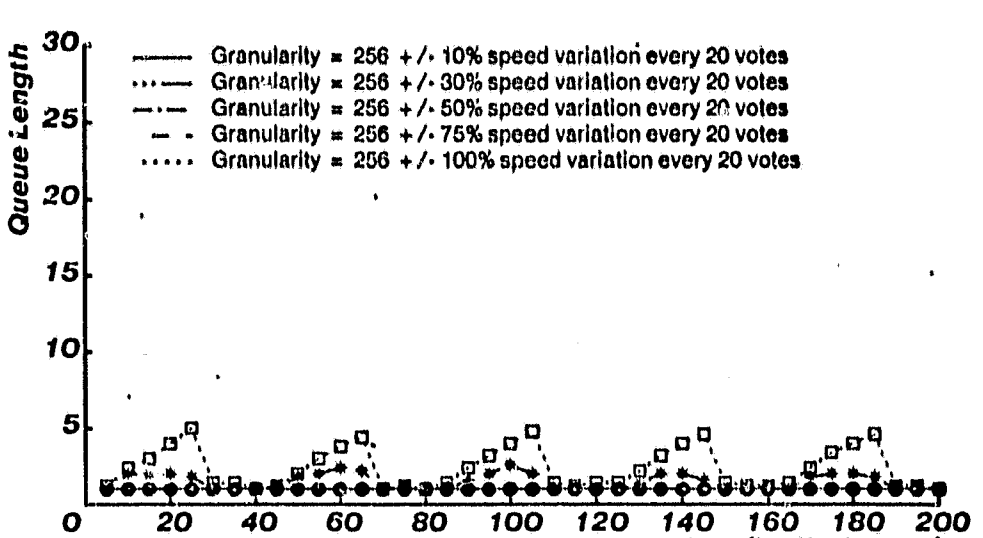

Figure 1-8: Granularity equal to 256, one subtask slower half the time, faster half the time

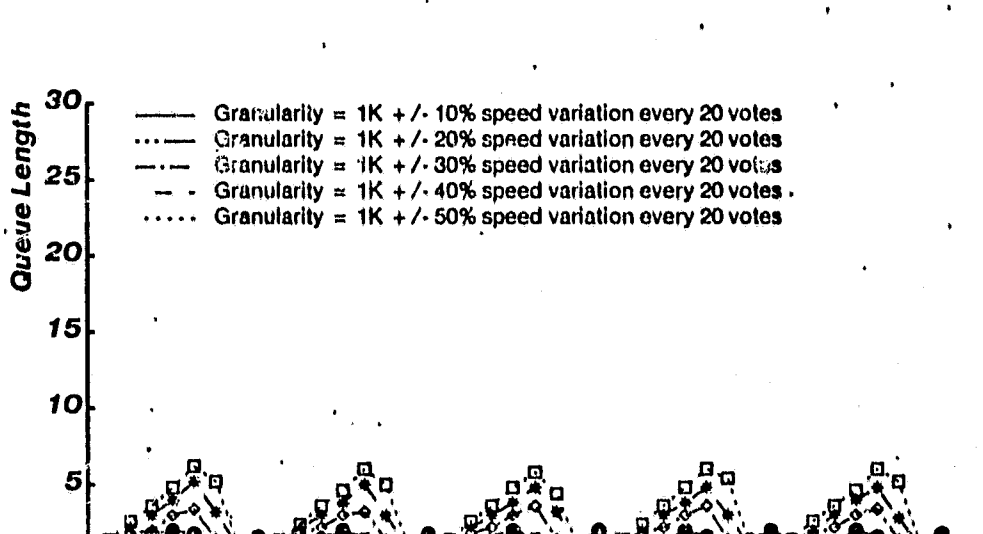

Figure 1-9: Granularity equal to 1024, one subtask slower half the time, faster half the time

always takes as long to calculate a new data value as the others? As soon as a subtask falls behind, it no longer pays the overhead cost, since it has messages queued up waiting for processing. This fact

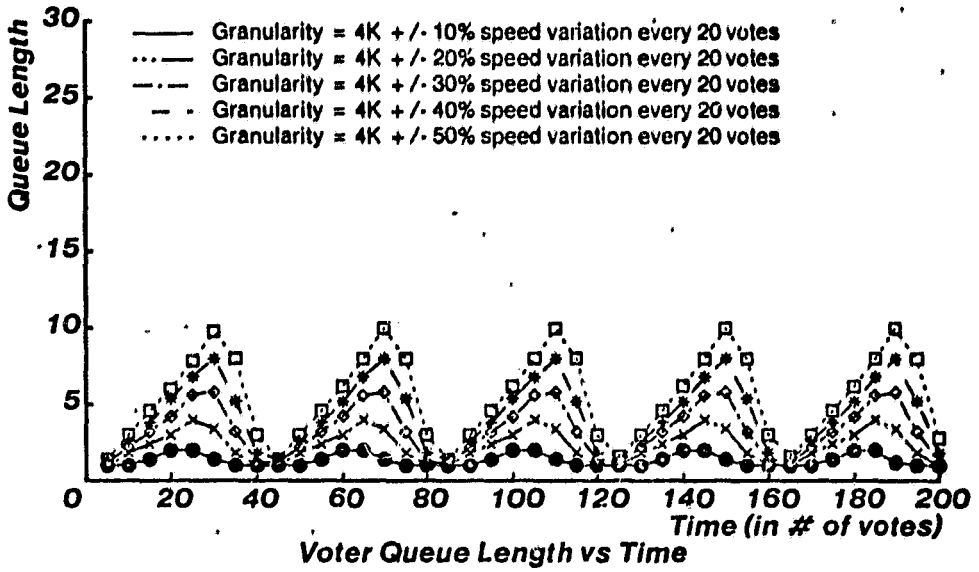

Figure 1-10: Granularity equal to 4K, one subtask slower half the time, faster half the time

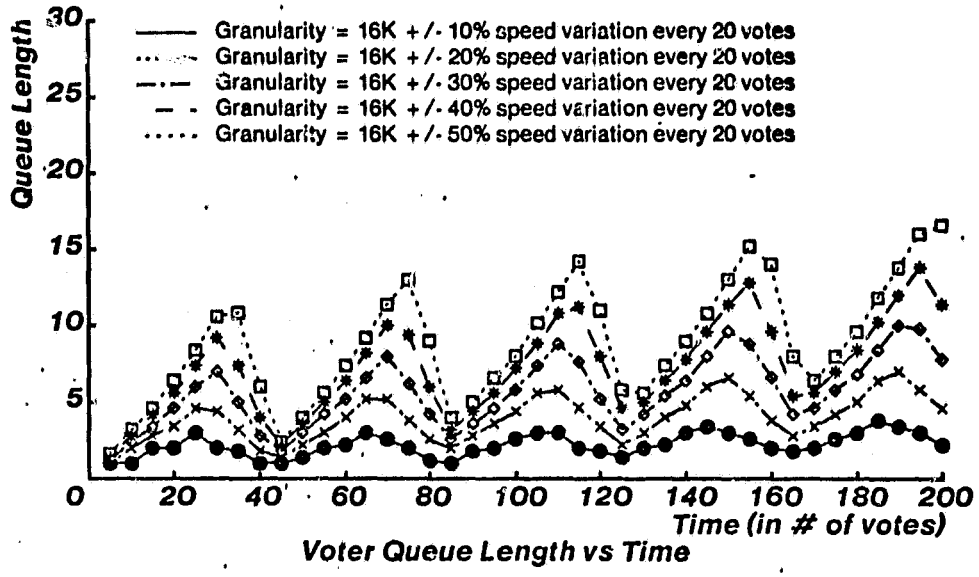

Figure 1-11: Granularity equal to 16K, one subtask slower half the time, faster half the time

would imply that a subtask can catch up, and the rate at which it catches up is the voter overhead cost per iteration.

The experiment can be described as follows: one subtask will do additional operations (10% to 50%) for 20 iterations followed by a period of normal behavior (performing the same number of operations as the other subtasks). The results of the experiment are shown in Figures 1-12 to 1-14. It can be seen that during the periods of normal operation for all three subtasks, the queue length declines, and given a long enough period of normal behavior would reach one. The rate of decline of queue length during normal subtask behavior indicates the effect of voter overhead on the subtasks.

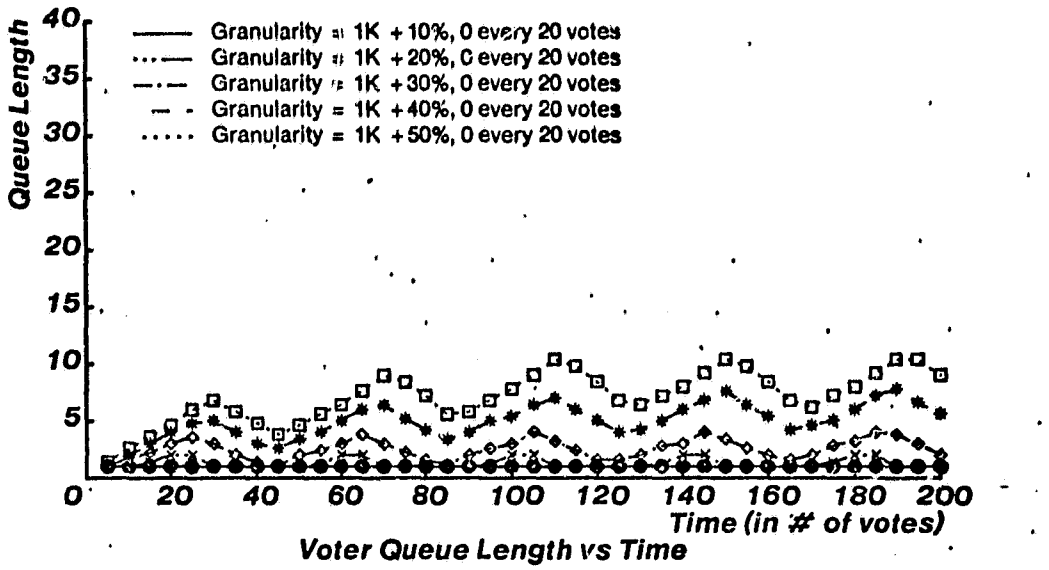

Figure 1-12: Granularity equal to 1024, one subtask slower half the time, same half the time

1.8. Experimental Analysis

The three experiments performed give a clear picture of a synchronization model for the equal subtasks paradigm. There appear to be two factors involved in the model. The factors are:

1. There is a minimum voter overhead that is due to the time required by the voter to receive a message, handle the data, and vote on the data. The subtasks that have a queue length of one must pay this overhead cost every iteration of the experiment. One might be encouraged to design a voter with very high overhead, in order to allow greater process speed variation.

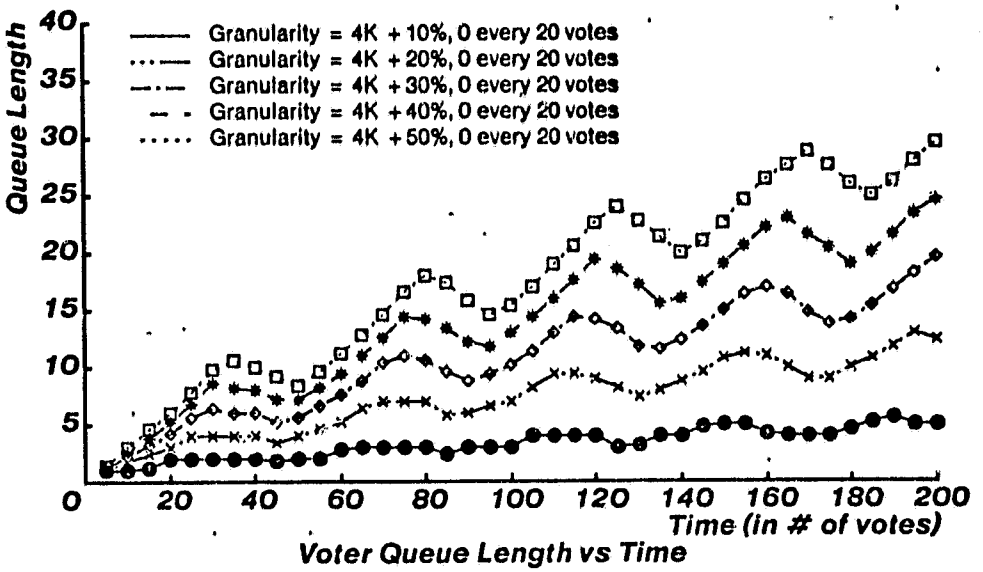

Figure 1-13: Granularity equal to 4K, one subtask slower half the time, same half the time

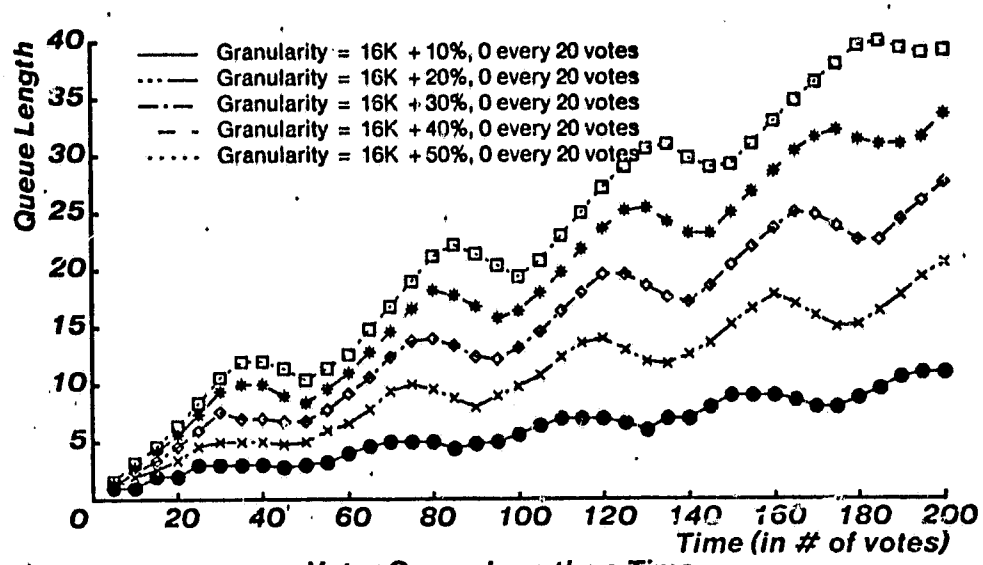

Figure 1-14: Granularity equal to 16K, one subtask slower half the time, same half the time

2. The overhead cost increases as the voter queue length increases due to an increase in the data handling cost. This factor would indicate that for a long enough queue, the voter

could mask any difference in process speed. For practical queue lengths, though, the increase in voter overhead masks only some of the subtask speed variation.

1.9. Conclusions

The synchronization experiments can give some design principles for TMR asynchronous voting systems. These principles can be applied to optimize the voter queue length, to choose a subtask granularity, and to determine the amount of process speed variation allowed in a design. Proper application of the principles will lead to a design that will have a bounded queue length for all possible variations in process execution rate. The principles can be summarized as follows:

1. Smaller granularity subtasks have a higher probability of having a bounded queue length.

2. As subtask granularity increases, the random variation in process speed becomes increasingly important in ensuring a bounded queue length.

3. A system that spends an equal amount of time being faster, and slower will have a bounded queue length.

4. Greater voter overhead allows a greater variation in process execution rate. This yields an interesting trade-off in voter design, since a faster voter process will increase system throughput, but will decrease the amount of variation permitted in process execution rate.

These results can be generalized for synchronous voting, as well as asynchronous voting. If the maximum voter length is fixed at one, then the system is synchronous like SIFT [9] [8] [6] and C.vmp [26] [19]. Both of these NMR systems use a synchronous voter with queue length of one. C.vmp has a hardware voter with a built in wait feature. The length of the wait corresponds to the voter overhead in these experiments. SIFT uses fixed scheduling, so a vote proceeds when the next time slot begins. The voter overhead corresponds to the design margin in the fixed schedule (the time between the end of the process execution, and the end of the time slot).

References

[1] Abraham, Jacob A., and Siewiorek, Daniel P.
An Algorithm for the Accurate Reliability Evaluation of Triple Modular Redundancy Networks.
IEEE Transactions on Computers :682-692, July, 1974.

[2] Apperson, Jerry L.
Bliss-11 Programmer's Manual
Digital Equipment Corporation, 1974.

[3] Castillo, Xavier.
Workload, Performance, and Reliability of Digital Computing Systems.
PhD thesis, Carnegie-Mellon University, December, 1980.

[4] Davies, Daniel and Wakerly, John F.
Synchronization and Matching in Redundant Systems.
IEEE Transactions on Computers C-27(6):531-539, June, 1978.

[5] Dolev, Danny.
The Byzantine Generals Strike Again.
Journal of Algorithms, December, 1981.
Stanford University.

[6] Forman, Phil and Moses, Kurt.
SIFT: Multiprocessor Architecture for Software Implemented Fault Tolerance Flight Control and Avionics Computers.
Third Digital Avionics Systems Conference :325-329, November, 1979.

[7] Frison, Steve and Wensley, John.
Interactive Consistency and Its Impact on the Design of TMR Systems.
In *12th Annual International Symposium on Fault Tolerant Computing*, pages 228-233. IEEE Computer Society, June, 1982.

[8] Goldberg, Jack.
The SIFT Computer and Its Development.
1980.
SRI International.

[9] Goldberg, Weinstock, Green, Kautz, Lamport, Melliar-Smith.
Development and Evaluation of a SIFT Computer: SIFT Operating System.
Interim Technical Report 2, SRI International, April, 1980.

[10] Goldberg, Jack.
The SIFT Approach to Fault Tolerant Computing.
1981.
SRI International.

[11] Hecht, H.
Reliable Software for Spacecraft.
In *Proceedings of Compcon*, pages 143-146. IEEE Computer Society, 1980.
Spring.

[12] Jones, Anita K., and Gehringer, Edward F.
The Cm Multiprocessor Project: A Research Review.*
Technical Report CMU-CS-80-131, Carnegie-Mellon Univerisity, July, 1980.

[13] Kong, Thomas H.
Measuring Time for Performance Evaluation of Multiprocessor Systems.
Master's thesis, Carnegie-Mellon University, November, 1982.

[14] Kuehn, Ralph E.
Computer Redundancy: Design, Performance, and Future.
IEEE Tranactions on Reliability R-18(1):3-11, February, 1969.

[15] Lala, Jay H., and Smith, Charles J.
Performance and Economy of a Fault-Tolerant Multiprocessor.
In *1979 Proceedings of the National Computer Conference*, pages 481-492. National Computer Conference, 1979.

[16] Lamport, Leslie.
Time, Clocks, and the Ordering of Events in a Distributed System.
Communications of the ACM 21(7):558-565, july, 1978.

[17] Malaiya, Yashwant K., and Su, Stephen Y. H.
A Survey of Methods for Intermittent Fault Analysis.
In *1979 Proceedings of the National Computer Conference*, pages 577. National Computer Conference, 1979.

[18] McConnel, Stephen R., and Siewiorek, Daniel P.
CMU Voter Chip.
Technical Report CMU-CS-80-107, Carnegie-Mellon University, March, 1980.

[19] McConnel, Stephen R., and Siewiorek, Daniel P.
Synchronization and Voting.
IEEE Transactions on Computers C-30(2):161-164, February, 1981.

[20] Melliar-Smith, P. M., and Schwartz, R. L.
Hierarchical Specification of the SIFT Fault Tolerant Flight Control System.
Technical Report, SRI International, 1980.

[21] Michalopoulos, Demetrios A.
Uniquely Maneuverable Fighter Plane to Use Digital Processors.
Computer, October, 1982.

[22] Musa, John D.
Software Reliability Measures Applied to System Engineering.
In *1979 Proceedings of the National Computer Conference*, pages 941-946. National Computer Conference, 1979.

[23] Ousterhout, John K., Scelza, Donald A., and Sindhu, Pradeep S.
Medusa: An Experiment in Distributed Operating System Structure.
Communications of the ACM 23(2):92-105, February, 1980.

[24] Pease, M., Shostak, R., and Lamport, L.
Reaching Agreement in the Presence of Faults.
Journal of the ACM 27(2):228, April, 1980.

[25] Segall, Singh, Snodgrass, Jones, Siewiorek.
An Integrated Instrumentation Environment for Multiprocessors.
1982.
Carnegie-Mellon University.

[26] Siewiorek, Kini, Mashburn, McConnel, and Tsao.
A Case Study of C.mmp, Cm*, and C.vmp: Part 1 - Experiences with Fault Tolerance in Multiprocessor Systems.
Proceedings of the IEEE 66(10):1178-1199, October, 1978.

[27] Siewiorek, Daniel P. and Swarz, Robert S.
The Theory and Practice of Reliable System Design.
Digital Press, Bedford, Mass., 1982.

[28] Siewiorek, Daniel P., Bell, C. Gordon, and Newell, Allen.
Computer Science Series: Computer Structures: Principles and Examples.
McGraw Hill, 1982.

[29] Sindhu, Pradeep and Singh, Ajay.
Performance Evaluation of Message Mechanisms.
Carnegie-Mellon University.

[30] Singh, Ajay.
Pegasus: A Controllable, Interactive, Workload Generator for Multiprocessors. .
Master's thesis, Carnegie-Mellon University, December, 1981.

[31] Sklaroff, J. R.
Redundancy Management Technique for Space Shuttle Computers.
IBM Journal of Research and Development :20-28, January, 1976.

[32] Snyder, F. G.
A Comparison of Redundant Computer Configurations.
In *Proceedings of Compcon*, pages 125-133. IEEE Computer Society, 1980.
Spring.

[33] Wensley, Lamport, Goldberg, Green, Levitt, Melliar-Smith, Shostak, and Weinstock.
SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control.
Proceedings of the IEEE 66(10):1240-1255, October, 1978.

[34] Yemini, Yechiam and Cohen, Danny.
Some Issues in Distributed Processes Communication.
In *Proceedings of the First International Conference on Distributed Computing Systems*, pages 199-203. IEEE, October, 1979.
Huntsville, AL.

ABSTRACT

The trend towards integration of avionics in flight controls in future aerospace systems requires an ever increasing complexity in the on- board computing systems. NASA Langley Research Center has created an Avionics Integrated Research Laboratory (AIRLAB) as a facility for developing the methodology for integrating avionics in flight controls. Due to the complexity of these systems, extensive testing will be required to validate that the system hardware and software function according to specification. Engineering prototypes for two fault tolerant multiprocessors--SIFT (Software Implemented Fault Tolerance) and FTMP (Fault Tolerant Multiprocessor)--have been delivered to AIRLAB.

The goal of this research was to define experiments that can be used to validate fault free performance of multiprocessor systems. These experiments were refined through implementation on the Cm* multiprocessor testbed at Carnegie-Mellon University. Future research will adapt and modify these experiments for FTMP and/or SIFT.

1. Objectives

The National Aeronautics and Space Administration (NASA) has ongoing research into the integration of avionic and control functions for aircraft in the 1990-and-beyond time frame. As a focus for this technology, NASA Langley Research Center (LARC) has established an Avionics Integrated Research Laboratory (AIRLAB). The goals of AIRLAB are to [1]:

1. develop the technology and methodology required to integrate avionic and control functions for aircraft

2. evaluate and study candidate system architectures

3. validate implementation technologies

4. establish a data base of performance, reliability, and experimental statistics.

The benefits to be derived from AIRLAB include:

1. definition and assessment of advance avionic system concepts including high reliability, fault tolerance, and effective maintenance

2. development of a credible data base for industry including systematic definition of system concepts, a catalogue of alternative features, and a methodology for design evaluation and design trade-offs

3. demonstration of experimental systems.

Computers on-board current jet transports perform isolated functions, are usually of simple architectures, and are not flight critical. If a computer fails, the flight crew can assume the function formerly done by the computer. In the Aircraft Energy Efficiency (ACEE) Program, NASA studied the design of innovative aircraft which reduce fuel consumption. Operating with reduced stability margins, these aircraft require active computer control. These computers must have a reliability comparable to other aircraft subsystems. A goal of 10^{-10} failures per hour has been set.

In order to meet the active flight control and reliability requirements, complex computer structures have evolved. NASA Langley Research Center has contracted engineering prototypes of two multiprocessor architectures: SIFT (Software Implemented Fault Tolerance) [2] conceived by SRI International and fabricated by Bendix Corporation; and FTMP (Fault Tolerant Multiprocessor) [3] conceived by MIT's Charles Stark Draper Laboratory, Inc. and fabricated by Collins. The engineering

prototypes for both SIFT and FTMP have been delivered to the AIRLAB facility.

The goal of the research is to define a set of tasks that can be used to provide a demonstration of the fault free performance of SIFT and/or FTMP. Here we take the meaning of performance in its broadest sense to include functionality and speed.

2. Background

Digital computer systems are enormously complex. In order to make them easier to comprehend, it is necessary to divide the system into several levels [6]. One can then proceed from the most primitive level upwards to the highest conceptual level by introducing a series of abstractions. Each abstraction contains only information important to its particular level, and suppresses unnecessary information about lower levels. The levels in a digital system frequently coincide with the system's physical boundaries since the concept of levels was utilized by the system's designers to manage complexity. Once details at one level are comprehended, only the functionality provided for the next higher level need be considered. Figure 1 depicts one possible set of levels of abstractions.

Level	Sublevel	Typical Components
Multiprocessor		Processor, memory, switches
Program	Application Software	Display, navigation, flight control
	Executive Software	Message system, task scheduler, memory allocator
	Instruction Set	Memory state, processor state, effective address calculation, instruction execution
Hardware	Logic	Gates, flip-flops, registers, sequential machines

Figure 1. Levels of Abstraction in Multiprocessor Systems

AIRLAB is a facility for testing and measuring fault tolerant architectures. Our experience at CMU indicates multiprocessors go through an evolution of stages. A stage is defined by the amount of functionality available to the user. This functionality, in turn, determines the complexity and sophistication of experiments that can be run.

There are several activities in the life of an experiment. First, the code has to be designed and written. Next, it must be compiled, followed by loading, debugging, measurement, and analysis. Another view of the stages of a system's life is the number of these activities that are directly supported by the system for the user.

The following are three representative stages in the evolution of a system.

2.1. Stage 1 - Standalone

The system is completed through the instruction set level of abstraction. That is, the instruction set has been defined and the hardware has been implemented. There is virtually no software to support user applications. The only software utility would be a loader whereby programs compiled on another machine can be loaded into the system under test. Experiments are limited to simple, regular, compute bound algorithms. Only a limited number of parameters may be varied, and this variation requires rewriting of the source code of the experiment. There are several attributes to Stage 1 experiments. The programmer must be a hardware expert since there is little software to provide a higher level virtual (abstract) machine. Hence the program is tied closely to the hardware. The user must specify where code is placed, define the memory map, and write code to initialize the memory, create processes, manage resources, and collect data.

Typical experiments in Stage 1 include:

- **Hardware Saturation.** Programs consist of two or three instruction loops with variation in placement of code and data. The capacity of various system hardware resources is determined as well as the impact of contention for those resources.

- **Speedup due to Algorithm/Data Variation.** Experiments seek the impact of synchronization for data, as well as variation due to data dependencies and size of data.

- **Errors.** Diagnostic programs can be continuously run and monitored on the system. Distribution of diagnostic detected errors can be studied.

2.2. Stage 2 - Operating System

The user is presented the abstraction provided by the executive software. This software provides basic functions such as resource management and scheduling. In programming experiments, the user is employing operating system primitives. Hence, the user needs a substantial operating system expertise. Also, characteristic for this phase is the discrete incremental nature of the experimentation process; each experiment represents one point in the design space.

The attributes of Stage 2 applications can be stated as follows:

- very regular, data bound with limited variation of parameters
- the general program organization has a Master process controlling a collection of Slave processes doing the actual computation
- code is replicated
- heavy use of OS mechanisms

Typical experiments are:

- **Measurements of the cost-per-feature of the operating system's functions.** Experiments exercise statically each OS function on a one by one basis. Examples include: memory management, communication primitives, synchronization, scheduling and exception handling.
- **Measurements of different implementation of parallel algorithms.** The impact of using various strategies in parallel program organization, data structure and resource allocation is studied.

2.3. Stage 3 - Integrated Instrumentation Environment

At this stage hardware and software have been provided for generating experimental stimulus, dynamically observing hardware and software activities, and analyzing results. With this enhanced support, the user can experiment at the application level of abstraction with full variation of parameters. A major characteristic of this stage is the provision of stimulus generation, monitoring, data collection and analysis grouped under a unique user interface. Also the OS, the support software and the user application are uniformly instrumented enabling improved behavior visibility. Only with this capability, the interaction between OS, support software and user application became measurable with acceptable effort. Hence, the programmer could be a relative system novice. Experiments at this stage have the following attributes:

- Measurements of dynamic behavior of OS and applications.
- Measurements are continuous. Program could be monitored on-line and sometimes in real-time.
- Studies of different virtual machines.

- Studies of different logic intercommunication structures.

- Scaling application performance with respect to different virtual machines.

Examples of experiments at this stage include:

- Comparison of various OS policies as reflected by classes of applications.

- Tuning a virtual machine for a specific application.

- Designing application oriented architectures.

- Study of multiprocessor intercommunication strategies.

- Validation of fault-free performance of an emulated system.

3. Proposed Experiments

For purposes of experimental transfer, the target AIRLAB system will be assumed to be equipped with Stage 2 software (i.e., software through the executive level of abstraction with the ability to load a program and for the program to write a file of experimental data). All these experiments were conducted on Cm*, CMU's 50 processor multiprocessor system. Thus, experiments have been attempted and scientific questions formulated prior to implementation at AIRLAB. These experiments were up to and including executive software level of abstraction. No assumptions were made about constraints at the application level of abstraction which would limit the utilization profile of the lower levels. Performance and logical limits of individual executive and hardware functions were explored. Points where the system saturated or ceased to perform to specification were sought and documented. Each of these individual dimensions are dynamically stressed by an application. With the limits of performance documented, we can intelligently select application level experiments that are more likely to stress multiple dimensions in a way that the application may cease to meet its specifications. A summary of suggested experiments and supporting references follow.

- **Baseline non fault tolerant system reliability**

  Purpose: To derive a baseline of non-redundant hardware and/or software system reliability.

  Description: It is assumed that a file has been created with information on system crash behavior and/or errors detected from execution of diagnostics. This file will be analyzed to determine Mean Time To Failure and

Mean Time To Error. If data is sufficiently detailed, it may be possible to develop a mathematical model that fits the data.

- **Determination of execution speed of hardware**

Purpose: To determine the variation in execution speed between different processors and to produce a table for normalizing the measurements done on different processors. Speed differential is a bound on the accuracy of performance related experiments.

Description: Each processor performs an identical task over a sufficiently long period so that the results are repeatable.

- **Telling time in a multiprocessor**

Purpose: As time is essential in monitoring and measuring an experiment, the overhead to tell time and limits of accuracy of telling time must be documented.

Description: Both single and multiple clocks will be used. The use of a single clock insures uniformity of absolute and relative (differential) time throughout the multiprocessor provided the clock reading software adds only a small, constant delay. If time is required frequently or if reliability is a consideration, multiple clocks can be used. This experiment will measure variability and contention for reading a single clock as well as measuring the differences in multiple clocks to determine clock drift.

- **Communication mechanisms**

Purpose: To determine the maximum information transmission rate of a hardware/operating system combination. This is an upper bound on information flow within the multiprocessor.

Description: This experiment will determine the execution time for sending messages as a function of the number of bytes transferred. Various forms of message sending will be measured. These will include mechanisms provided by the executive software as well as mechanisms that can be programmed by a user.

- **Operating system calls**

Purpose: Each operating system service (or call) adds overhead to an

application.

Description: This experiment will measure the overhead as a function of call type, call frequency, contention, and relative positioning of code. In SIFT, particular attention will be paid to the software voting mechanism.

- **Impact of time skew**

Purpose: Due to the delays and overheads measured in the above experiments, it will not be possible to keep the multiple, asynchronous copies of application code in identical lock step.

Description: This experiment will introduce variable time skews into redundant code to determine if or when the multiple copies of the application get out of logical sequence and causes the system to cease functioning.

- **Validation of instruction set architecture**

Purpose: To determine whether automatically generated diagnostic programs have sufficient fault detection coverage to be run periodically and alleviate the problem of fault latency.

Description: Software developed in [7] takes a formal description of a computer instruction set and then generates a program which tests that instruction set. This methodology outperformed manufacturer supplied diagnostics for faults that were inserted at the instruction set level. These automatically generated programs contained a factor of 20 fewer instructions than the manufacturers' diagnostics while achieving better coverage (i.e., 98.5% vs. 95.5%).

The appendices contain details of all but the first and last experiment as they were conducted on Cm*. Results of the first experiment can be found in [5] while results of the last experiment were reported in [7].

4. References

[1] Research Triangle Institute, Systems and Measurements Division, "Validation Methods Research for Fault-Tolerant Computer Systems," Preliminary Working Group II Report for Langley Research Center, National Aeronautics and Space Administration, September 7-8, 1979.

[2] Wensley, J. H., L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak and C. B. Weinstock. "SIFT:

Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proc. IEEE*, vol. 66, no. 10, October 1978, 1240-1255.

[3] Hopkins, A. L., Jr., T. B. Smith, III and J. H. Lala, "FTMP--A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft," *Proc. IEEE*, vol. 66, no. 10, October 1978, 1221-1239.

[4] Bell, C. G. and A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill Book Co., New York, NY, 1971.

[5] Siewiorek, D. P., V. Kini, H. Mashburn, S. McConnel and M. Tsao, "A Case Study of C.mmp, Cm*, and C.vmp: Part I--Experiences with Fault Tolerance in Multiprocessor Systems," *Proc. IEEE*, vol. 66, no. 10, October 1978, 1178-1199.

[6] Siewiorek, D. P., C. G. Bell and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill Book Co., New York, NY, 1982.

[7] Lai, K. W., "Functional Testing of Digital Systems," Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, December 1981.
1199.

Measuring Time in Multiprocessor Systems

by
Thomas H. Kong, Alfred Z. Spector, Daniel P. Siewiorek

29 November 1982

Abstract

A system clock is often used as a time-keeping device for measuring software performance. However, in a distributed system where there is only one system clock, it may be difficult to obtain accurate clock readings. This is because of communication delays and clock contentions. This report investigated the problems associated with a single time base in multiprocessor systems. The multiprocessor Cm* was used as the research vehicle. First, the accuracy of the clock was found to be greatly affected by the number of simultaneous clock reads as well as the overall system workload. Second, methods were developed to compensate the clock readings by monitoring the system load during the time measurements. The accuracy of these methods was better than 7μS. Finally, an experiment was performed to measure the latency of messages and the execution time of message-based remote procedure calls.

This project was supported by NASA Langley Research Center under contract number NAG-1-190, by NSF under contract number MCS-8120270, and by the Department of the Army under contract number DASG-60-80-C-0057.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NASA, NSF, the Department of the Army, or the U.S. Government.

Table of Contents

List of Figures

1 Introduction

While performance evaluation of computer hardware is commonly done with special hardware such as oscilloscopes and logic analyzers, performance evaluation of computer software such as operating systems can be done with software methods. The software method has the following advantages:

- It can be completely automated from data collection to data reduction.
- It can be performed remotely without accessing the internals of the machine.
- It requires only a system clock of good resolution.

Hardware methods can also be completely automated, but the measuring hardware may be rather complex. The disadvantage of software methods is that they are less accurate than hardware methods because of the possible interaction between the measuring software and the measured software.

To measure the performance of software, the system clock is often used to measure elapsed times. However, it is often difficult to obtain accurate clock readings from a system clock. This is because the amount of time required to read the system clock is variable due to system load, page traffic, etc. Therefore when a result is returned to the calling program, it may be inaccurate or outdated. In a distributed system where communication delay is dependent on system activity, and where a large number of subsystems attempting to read the clock causes contention, the result returned to the calling program may be even less accurate.

This problem is serious in the case of the Cm* [3] clock. A preliminary study showed that the result of a clock read can be erroneous by as much as 2mS, depending on the system load and the amount of contention for the clock. A goal of this project is to develop methods to read the clock more accurately so that software methods can be more widely used to measure system performance.

The purpose of this project is to investigate the feasibility of measuring the performance of multiprocessor operating systems using software methods.

The following section presents the background information for this project. Previous work

related to this project is surveyed and the research vehicle, Cm*, is briefly described. Section 3 discusses the problem with global clocks. The clocks in Cm* are described, and mechanisms for accessing them under different operating systems are presented. The clock reading software is examined and its performance as a function of load is studied.

Section 4 discusses elapsed time measurements on Cm*. Based upon the experience developed in using the Cm* clocks, methods are designed to yield more accurate results for elapsed time measurements. The accuracy of these methods is illustrated through tests.

An example of the usage of one of the methods developed in Section 4 is presented in Section 5. The example measurement is concerned with the latency of message mechanism and the execution time of message-based remote procedure calls. Finally, Section 6 presents the general conclusions.

2 Background

2.1 Previous Work

There has been a significant amount of work on performance evaluation of multiprocessors. Most of this work relied upon hardware devices for measuring time. When Raskin performed measurements on Cm* [10], he used the Cm* Map-Bus Monitor, logic analyzers and hardware counters. Marathe suggested that measurement tools should match the level of the measurement [6]. When he was measuring the operating system kernel performance of C.mmp/Hydra, he used both hardware and software methods. Snodgrass has also studied the problem of monitoring distributed systems [12] [13].

While all these works required the use of special system dependent hardware, this project aims at providing software methods for performance evaluation that require no special hardware other than high resolution system clocks. To utilize the system clocks for performance evaluation, the concept of time in a distributed system must be understood. Studies concerned with the understanding of time in a distribute system and the dissemination of system time have been made by Lamport [5] and Ellingson [1].

2.2 Research Vehicle

The research vehicle used in this project is Cm* and its two operating systems - StarOS and Medusa. In this subsection, a brief overview of the research vehicle is presented. More detailed descriptions can be found in the research review edited by Jones and Gehringer [3].

2.2.1 Cm* Hardware Structure

Cm* is a multiprocessor consisting of fifty processor-memory pairs made up of DEC LSI-11's. Each processor-memory pair is called a *computer module*. These computer modules are grouped into five *clusters*, forming a hierarchical switching structure. The lowest level of the switching hierarchy consists of the *Slocals*, which are switches placed between each processor and its local memory. Their function is to determine if references generated by the processor can be directed to the local memory. If the references cannot be directed to local memory, the Slocal will forward the address through the *Map Bus* to the *Kmap* of that cluster for further address translation. The Kmap is a high speed microprogrammable communication controller. It provides the mechanism for the computer modules (Cm's) of its own cluster to communicate with each other, and it cooperates with other Kmaps to service the communication requests made to Cm's of other clusters. All communications between the Kmaps are implemented via packet-switching rather than by circuit-switching to avoid the possible deadlock over dedicated circuit-switching paths. In addition, since the Kmap is much faster than the main memory of the LSI-11's, the Kmap is active only for a small fraction of the time of a memory reference. Therefore, packet-switching allows the Kmap to service more than one request concurrently. Because of their microprogrammability, the Kmaps are also used extensively to implement key operating system functions of StarOS and Medusa.

2.2.2 StarOS

StarOS is a message based, object oriented operating system for Cm*. Its detailed description can be found in a technical report by Gehringer and Chansler [2]. Briefly, all StarOS information, including code and data, are contained in objects. Each object has an object type and a special set of operations defined for that object type. Users can also define their own abstract object types. A StarOS object is made accessible via the possession of a capability which contains the name of the object and a list of rights for that named object. Capabilities themselves do not contain the address information of the objects they name. Rather, they contain pointers to the *descriptors* which contain the physical locations of the objects. This way, if an object named by a number of capabilities is to

be relocated physically, only its descriptor needs to be updated while all the capabilities remain unchanged. The StarOS message facility supports the transmission of messages containing one capability or one data word. This implies that messages of size larger than one word are passed by reference. This pass by reference semantic is possible because names of objects are known system wide and a capability is sufficient to access an object anywhere in the system.

2.2.3 Medusa

Medusa is another message based operating system for Cm*. Its details were presented by Ousterhout *et al.* in [8] and in Ousterhout's thesis [9]. All Medusa information are stored in *objects* that are addressed through *descriptors.* The descriptors contain the type, the location, and the size of the objects. Descriptors are kept in protected objects known as *descriptor lists.* Each Medusa process, known as an *activity*, has two descriptor lists. The *private descriptor list* keeps the descriptors to objects that are private to the process, while the *shared descriptor list* keeps the descriptors to objects that are shared by all processes within the *task force.* A task force is defined as a collection of cooperating processes that perform a given computation. In Medusa, all objects are defined by the system and users are not allowed to define abstract object types. The message facility of Medusa supports messages of variable size. Messages are transmitted by value through special objects called *pipes* that are similar to the pipes of UNIX [11] in that they hold uninterpreted byte streams. The major difference from UNIX is that, in Medusa, only complete messages can be sent or received from the pipes, and that both the identity of the sender and the size of the message are available to the receiver.

3 Clocks in a multiprocessor

It was mentioned that due to communication delays, a clock read request is not received until sometime after the request has been made, and that the originator of the request does not receive the result until sometime after it has been transmitted. Therefore, the result of a clock read is often inaccurate. In this section, this problem will be examined in detail, and its effect on the performance of the clock on Cm* will be studied. Schemes designed to yield more accurate clock readings will be proposed.

A desirable solution to the problem of reading the clock is to have a globally readable clock with a communication delay that is small (compared to the clock resolution) and fixed regardless of

system load. Such a clock requires a special bus allowing multiple simultaneous read accesses for the broadcasting of the clock value. An example of such a bus structure is the interprocessor bus of C.mmp [15]. In the C.mmp implementation, there is a 56-bit global clock of 4 microseconds resolution. The value of this clock is continually broadcasted on the interprocessor bus.

However, in a more loosely coupled system, it is not feasible to devote a special purpose bus to the global clock because of the amount of cabling involved as well as the problems associated with bus arbitration over long lengths of wire. Also, broadcasting the clock value on the general purpose bus requires a large portion of the cycles available on the bus, thus significantly reduces the effective throughput of the bus for non-clock usage. For these two reasons, broadcasting the clock value is generally not done. Rather, the subsystem which needs to know the system time has to establish a connection with the clock and then to read its value. This way, communication occurs only when necessary. However, because the time required to establish a connection depends on bus activity and the transmission delay depends on the physical location of the subsystem, the total delay is unpredictable. When multiple requests for the system time arrive simultaneously, bus contention results and a queue is formed. The wait time in this queue adds further uncertainty to the total communication delay.

The conclusion is that global clocks require a special purpose bus which may not be feasible in a loosely coupled system. Without a special purpose bus, the accuracy of clock readings is sacrificed because of communication delays and contention.

3.1 Cm* clocks

Cm* currently provides three 32-bit real time clocks for time measurements. These clocks have a quartz crystal time-base with an adjustable resolution. The maximum resolution is 0.5 microseconds. The clocks can be zeroed under program control for interval measurements. Currently, all the clocks are hard-wired to give a resolution of 2 microseconds. This yields a maximum range of $2^{32} * 2\mu S = 2.386$ hours. The clocks are connected as peripherals to Cm3 on cluster 1, Cm4 on cluster 2, and Cm14 on cluster 5.

Since the LSI-11 uses memory mapped I/O, reading the clock is a simple read to a specific location in the I/O page (page fifteen) of the LSI-11 address space. For both StarOS and Medusa operating systems, reading the system clock is implemented via remote memory references.

3.2 Clock reading routines and their performance

In both StarOS and Medusa, clock reading is performed using procedure calls rather than by a LSI-11 "MOV" instruction. This is because the clock is 32 bits while the data bus is only 16 bits wide Thus, to read the full clock requires at least two memory references. Since the clock is always running, there is no guarantee that the high and low order words read correspond to the same 32-bit clock word. This is because after reading the first word, the low order word may overflow and wrap around at a clock tick, invalidating the first word read.

When this project began, both StarOS and Medusa provided a standard routine for reading the clock. For future reference, this algorithm is named "Varying-Read" algorithm because the clock register is read either three or four times depending on the value of the clock. Below is the pseudo-code for this routine:

```
Varying-Read:

FirstHi  = Read high order word of clock;
FirstLow = Read low order word of clock;
SecondHi = Read high order word of clock;
if SecondHi > FirstHi then begin
        SecondLow = Read low order word of clock;
        return SecondHi and SecondLow as the clock result;
end
else begin
        return FirstHi and FirstLow as the clock result;
end;
```

When SecondHi is greater than FirstHi, the low order word must have wrapped around between the first and second read of the high order word. Since it is not known whether the reading of FirstLow occurred before or after the wrap around, a second reading of the low order word must be taken.

A preliminary experiment was set up to evaluate the performance of the Varying-Read clock routine of Medusa. The objectives were to determine the average execution time of the routine and to see how the accuracy was affected by the system load. The experiment measured the elapsed time between two successive clock read procedure calls. This elapsed time was identical to the execution time of the routine including all the remote memory references.

The experiment was performed with eight Cm's distributed between clusters 2 through 5 reading the clock in cluster 1. The experiment was then repeated with thirty Cm's, also distributed between clusters 2 through 5. The results are summarized in Figure 1, which plots the elapsed time between two clock reads against the time elapsed since the beginning of the experiment. A simple calculation yields an intercluster memory reference rate of around 90 thousand references per second for the 30 Cm's case, and 53 thousand references per second for the 8 Cm's case.

Figure 1 reveals periodic peaks and troughs in the 30 Cm's curve. The peaks occurred when the low order word of the clock wrapped around during the execution of the second clock read routine, causing the second clock reading routine to read the low order word a second time. The troughs occurred when the low order word wrapped around during the execution of the first clock read routine, causing the low order word to be read again. For a detailed explanation of the reason for the peaks and troughs, refer to Kong's report [4]. Note that in the 8 Cm's case, there were fewer Cm's reading the clock and the probability of reading the clock while its low order word wraps around was much lower, hence the peaks and troughs did not appear regularly.

Figure 1 also shows that in the 30 Cm's case, the elapsed time rose from approximately 300μS to over 800μS and then fell from 800μS to approximately 300μS. This was because not all the Cm's started and finished simultaneously. Hence, there was less system load and contention at both the beginning and the end of the experiment, resulting in lower elapsed times. In the 8 Cm's case in Figure 1; the average value was approximately 300μS and no rise or fall was seen. This was because 8 Cm's reading the clock did not create sufficient traffic to slow down the clock read routines.

Because of the Varying-Read clock routine's erratic behavior when the low order word of the clock flips, two new clock reading routines were written. The first one was a modification to the original Varying-Read routine. It reads both the high order and the low order word twice, and has the property that it always returns the first low order word read as the low order word of the clock. Its execution time is essentially independent of the value of the clock readings, as shown in the performance diagram of Figure 2. In Figure 2 the rate of remote memory references for the 30 Cm's case was 96 thousand per second, while for the 8 Cm's case was 64 thousand per second. Below is the pseudo-code for the routine, which is referred to as the 4-Read clock routine because it always reads the clock register four times.

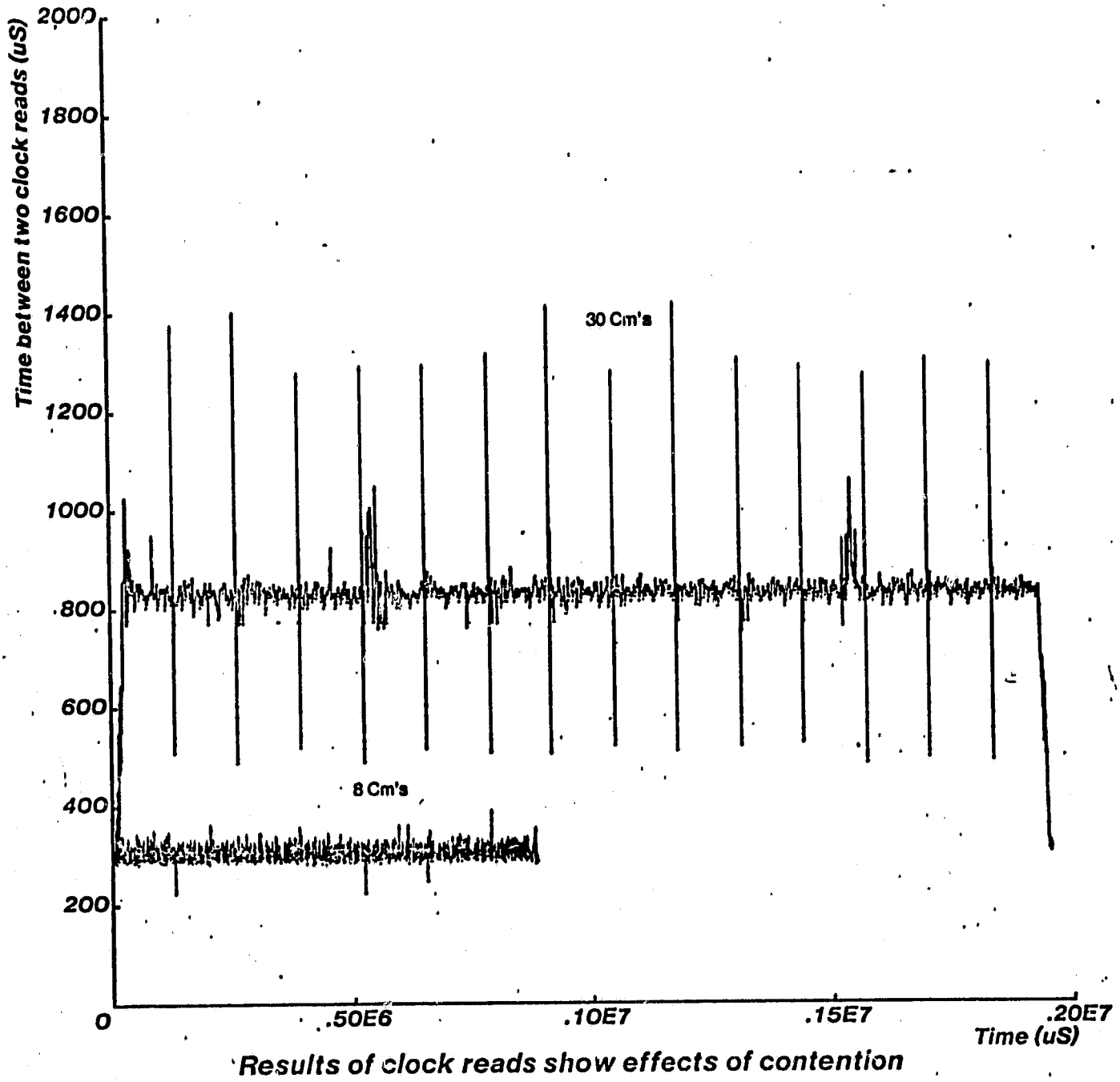

Figure 1: Performance of Medusa Varying-Read clock routine

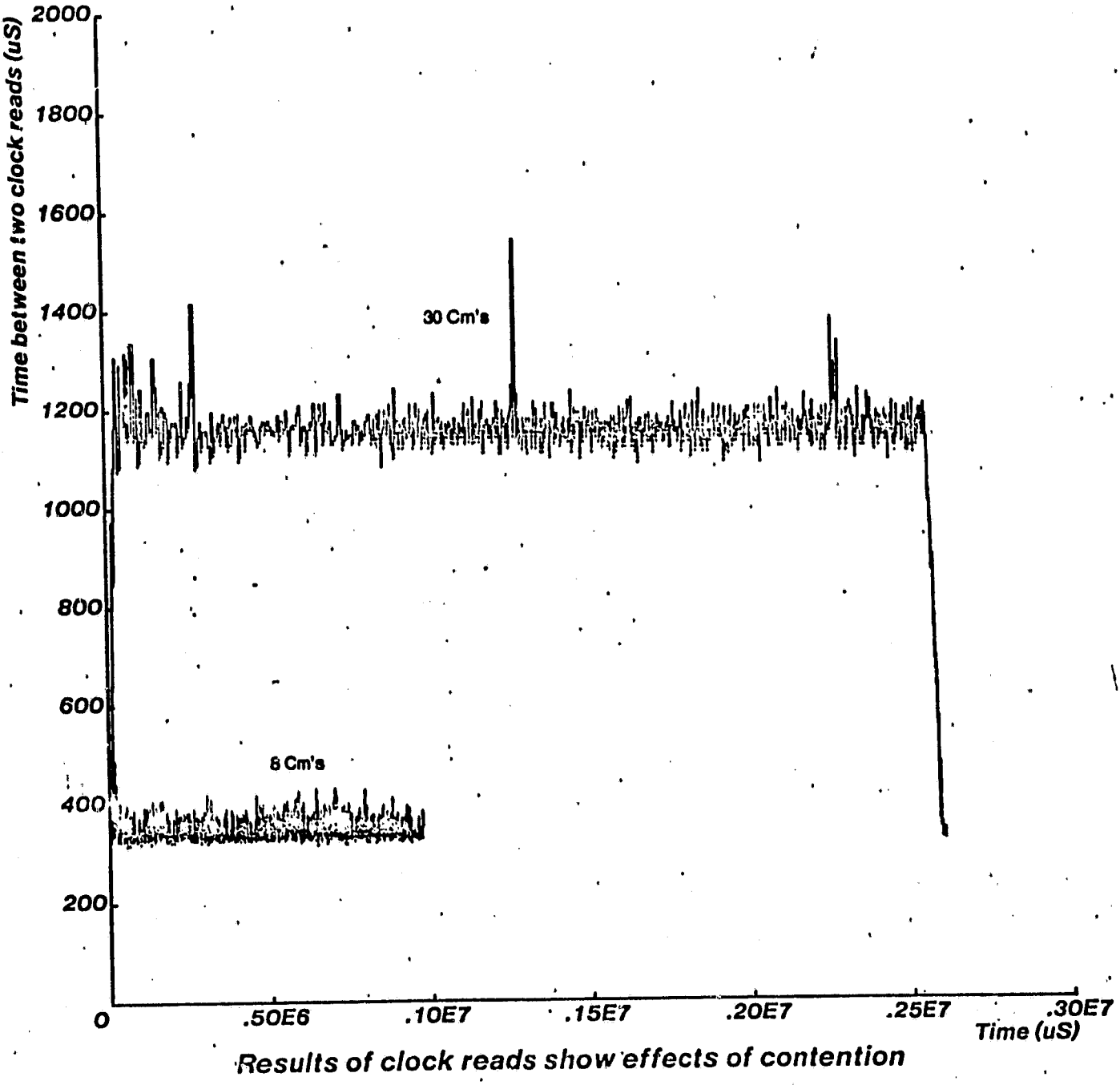

Figure 2: Performance of 4-Read clock routine running under Medusa

```
4-Read:

FirstHigh  = read high word of clock;
FirstLow   = read low word of clock;
SecondHigh = read high word of clock;
SecondLow  = read low word of clock;
IF SecondHigh > FirstHigh THEN          /* clock flipped */
        IF SecondLow.> FirstLow THEN    /* flip was before FirstLow */
                return FirstLow and SecondHigh as result
        ELSE    /* flip was after FirstLow read */
                return FirstLow and FirstHigh as result
ELSE    /* no flip occurred between FirstHigh and SecondHigh */
        return FirstLow and FirstHigh as result;
```

The second routine reads only the low order word of the clock and computes the value of the high order word. The routine makes use of two static variables $OldLow and $Hi. During a clock reset, these variables are zeroed. Every time the routine is called, the low order word of the clock is read and is compared with the value of $OldLow. Assuming the routine gets called at least once during the interval between two low order word flips, then if the value of $OldLow is higher than the current value of the low word, a flip must have occurred. The variable $Hi is then incremented. If the value of $OldLow is lower than that of the low word of the clock, no flip has occurred and the value of $Hi remains unchanged. This routine is called the 1-Read clock routine because it only reads the clock register once. Below is the pseudo-code for the routine:

```
1-Read:

STATIC $OldLow;
STATIC $Hi;

Low = Read low word of clock;
IF $OldLow >= Low THEN
        $Hi = $Hi + 1;
$OldLow = Low;
return Low and $Hi as the result;
```

This routine has an execution time that does not vary in time. The performance of this 1-Read clock routine is summarized in Figure 3. Here the remote memory reference rate was 76 thousand per second for the 28 Cm's case, and 23 thousand per second for the 8 Cm's case. The low rate of remote memory reference makes the routine execution time quite insensitive to the increasing number of Cm's reading the clock. Therefore, the differences between the 28 Cm's curve and the 8 Cm's curve were so small that the two curves overlap enough to be visually indistinguishable.

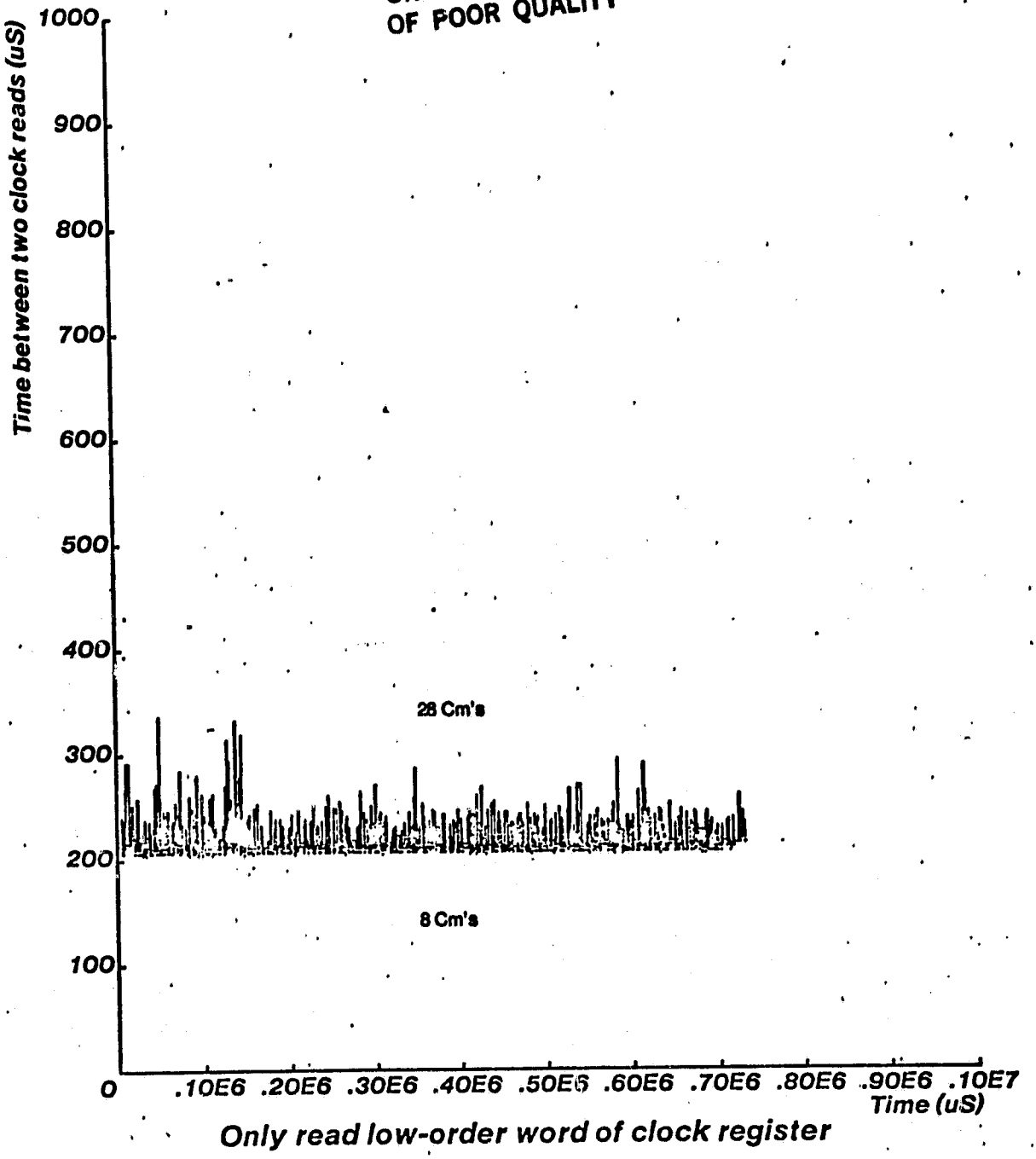

Figure 3: Performance of Medusa 1-Read clock routine

Note that this routine assumes the clock is read at least once in every interval T_f, where T_f is the time between two low word flips and is equal to $2^{16} * R$, where R is the number of seconds between

a clock tick. With the present R of 2μS, T_f equals 0.131 seconds. To use this routine, each Cm must have its own local copy of $Hi and $OldLow, and that each Cm must read the clock at least once in every T_f seconds.

Since the 1-Read and the 4-Read routines were to be used, a new set of experiments were set up to test the performance of both routines as a function of load under both StarOS and Medusa. The experiments involved the measurement of elapsed time between two clock reads as a function of the number of Cm's reading the clock.

3.2.1 StarOS results

The average execution time of the 4-Read clock routine increases with the increasing number of participating Cm's, and the standard deviation of the result also increases with increasing number of Cm's. Figure 4 is the summary of all the histograms normalized to give the same area under the curve. The distributions of all the curves appear to be Rayleigh with a lower bound of 310μS, which is the minimum time required to execute the 4-Read clock routine. A careful study of Figure 4 shows that beside the main peak, there is a small peak around 630μS to 760μS. This is due to interrupts[1] occurring between the two clock reads.

The average execution time of the 1-Read routine is quite insensitive to the increasing number of Cm's reading the clock. This is because the load generated by this clock routine is low enough that the Kmap can handle without saturating. Figure 5 is the summary of all the histograms normalized to give the same area under the curve. Since the average execution time varies very little and the standard deviation of the results remains almost constant, all the curves are similar and overlapping. A careful study of Figure 5 shows a small secondary peak around 470μS to 550μS. This is also due to interrupts by the line-time clocks.

3.2.2 Medusa results

The minimum time required to execute the 4-Read Medusa clock routine is approximately 320μS. The average time increases as the number of Cm's reading the clock increases. Figure 6 is the normalized plot of all the histograms. An interesting observation is that while the standard deviation of the result increases as the number of Cm's increases for small number of Cm's, it starts

---

[1] The line-time clock interrupts the processor sixty times per second.

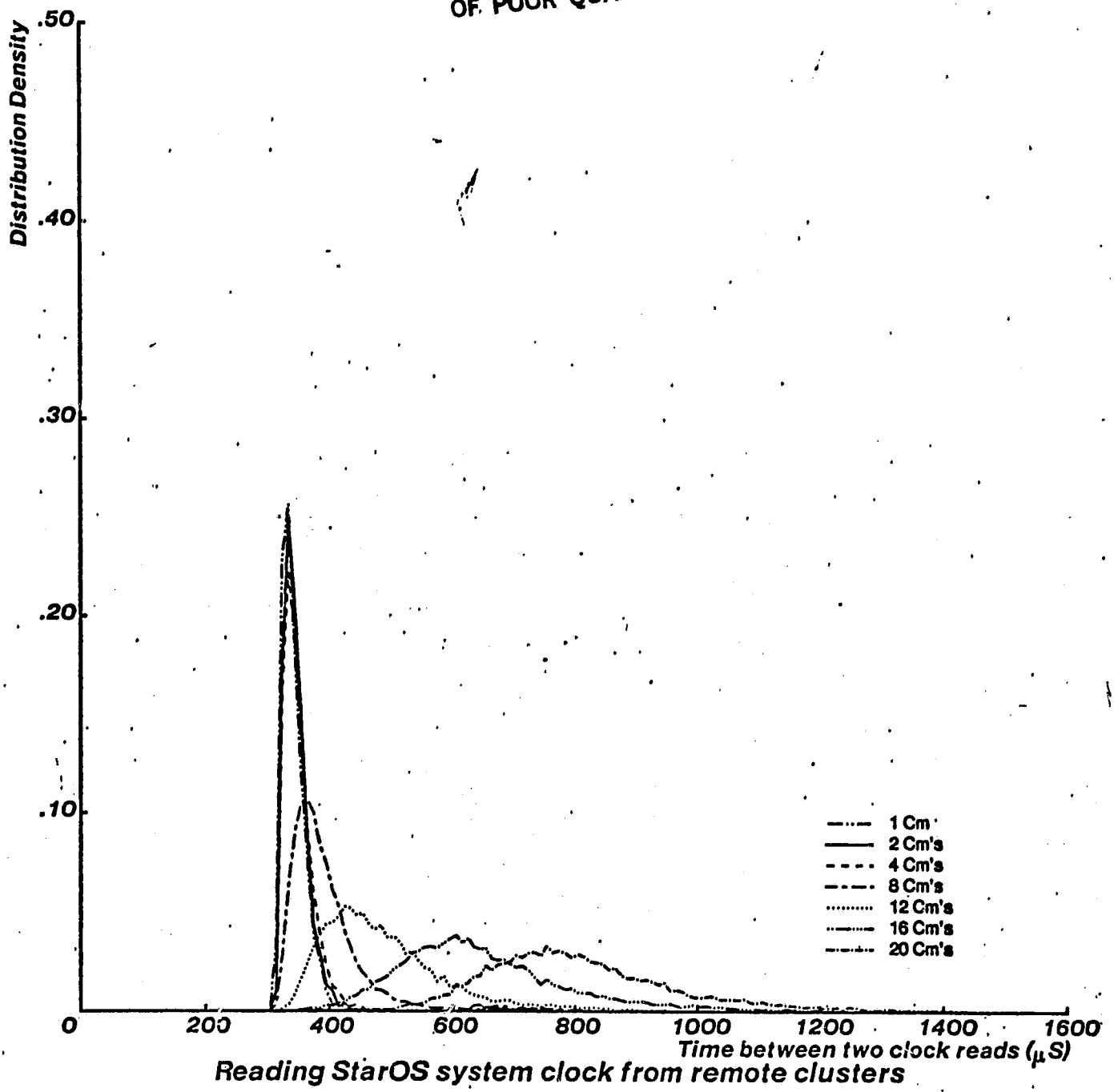

Figure 4: Performance of StarOS 4-Read clock routine

to fall at some point between 16 Cm's and 20 Cm's. This phenomenon is believed to be due to some complicated queuing mechanisms at the Kmap.

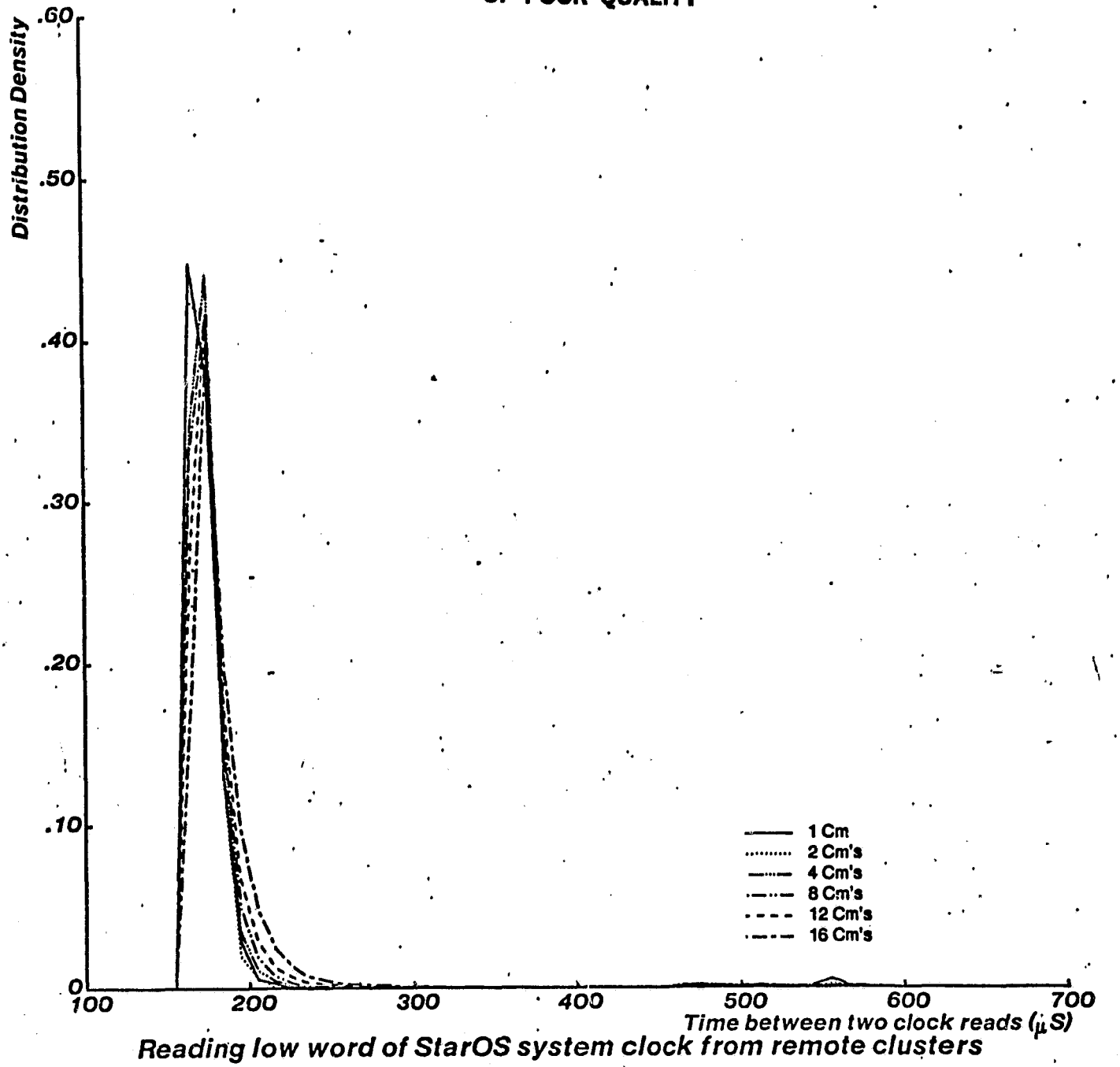

Figure 5: Performance of StarOS 1-Read clock routine

Figure 7 is the normalize distribution that summarizes the results of the Medusa 1-Read clock routine. The interesting point about this clock routine is that the standard deviation of the result is

extremely small, and that the average result does not change significantly with increasing number of Cm's reading the clock. This is because the load presented by this routine is so small that all clock read requests to the Kmap are processed immediately without having to wait in the Kmap queue.

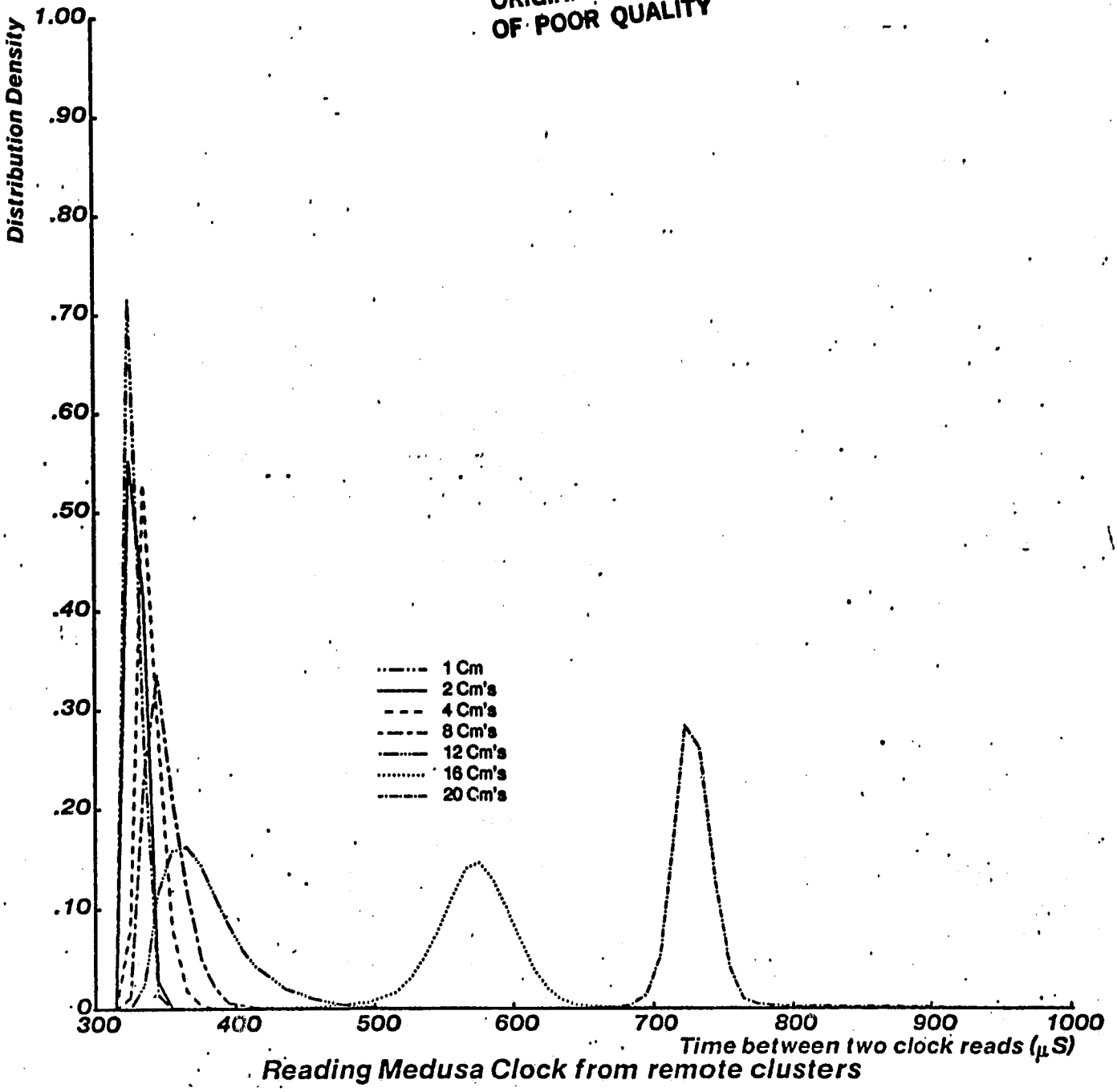

Figure 6: Performance of Medusa 4-Read clock routine

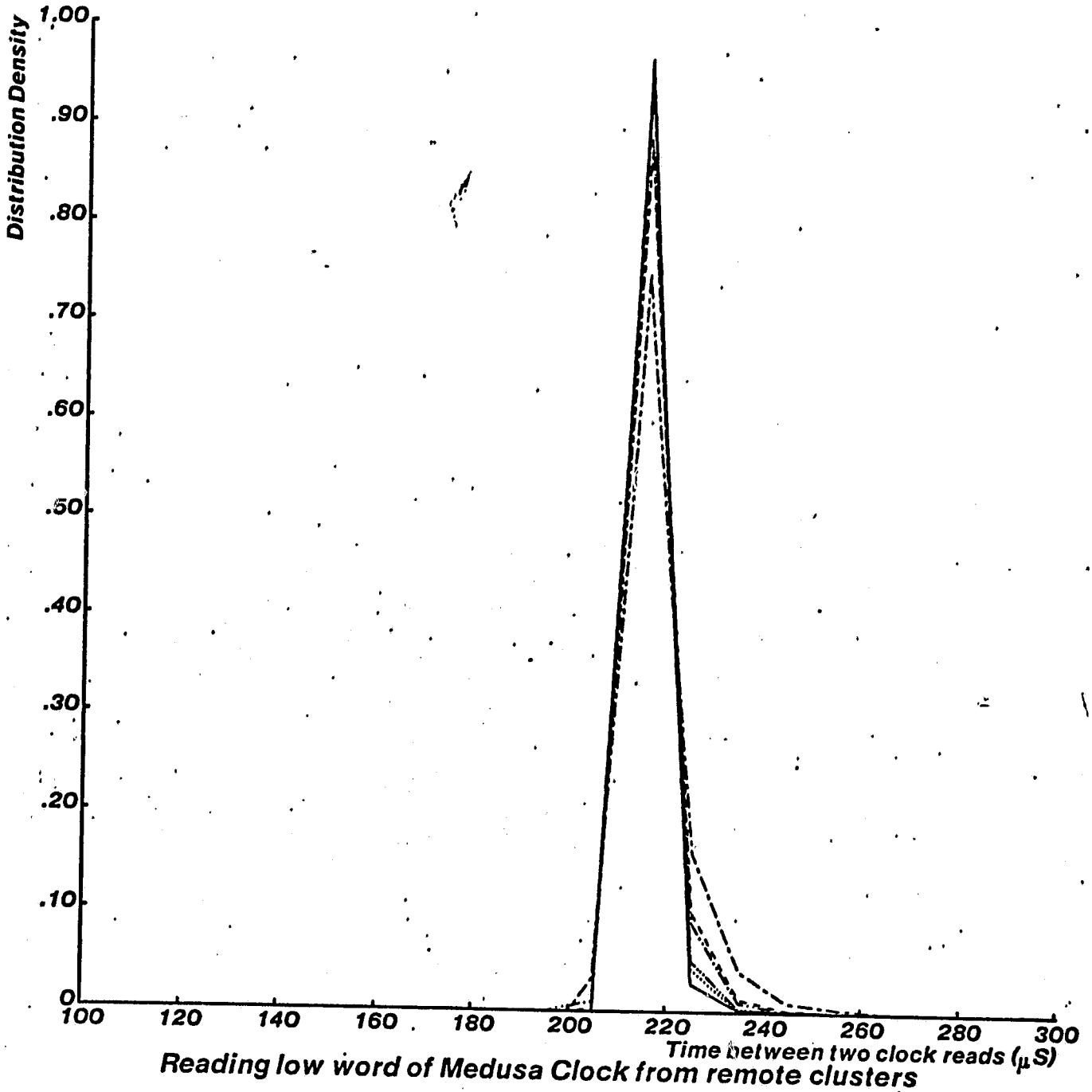

Figure 7: Performance of Medusa 1-Read clock routine

3.3 Conclusion

Comparing Figure 4 with Figure 6, one sees that even though the average execution times are roughly the same at light load, the execution time increases faster as a function of load under StarOS than under Medusa. The difference in the shape of the curves in Figure 4 and Figure 6 shows that the two operating systems have very different strategies for handling memory contention. Even when the effects of interrupts are ignored, the StarOS results show a larger standard deviation.

Although the Cm* global clock is capable of 0.5μS resolution, such resolution is not usable for accurate measurement of time intervals because of the uncertainty in delay involved in an intercluster reference in the presence of load. The results of the clock experiments show that short time intervals (500μS or less) cannot be accurately measured using any of the clock reading routines described.

One way to alleviate the problem is to read only the low order word of the clock. This way, only one LSI-11 instruction is needed to access the clock and the results should be much improved. The problem associated with just reading the low order word is the loss in clock range. With the clock tick set at 2μS, the range provided by the low order word is only about 0.131 second. Larger clock range can be obtained by increasing the clock tick value without sacrificing clock resolution because the usable clock resolution is limited by the uncertainty in communication delay. Therefore, a reasonable value for the clock tick should be commensurate with the uncertainty in communication delay. For example, under very light system load, reading a Medusa clock word has a standard deviation of 2.3μS, and reading a StarOS clock word has a standard deviation of 7.13μS[2]. Therefore, the 2μS resolution of the present clock is useful. However, under heavy loads, the standard deviation of reading a clock word can be very high. Under such loads, the clock tick can be lengthened substantially to increase the range of the clock without losing accuracy.

For any clock that cannot be completely read in one memory cycle[3], a clock read operation should be provided to latch the clock value and to allow all the clock words to be read indivisibly

---

[2] Assuming no processor interrupts.

[3] Because the clock register is wider than the data bus.

before another clock read operation is accepted. For Cm*, such a feature can be provided by using a hardware latch and some Kmap microprogramming. The clock read operation will latch the clock value, this value will then be moved indivisibly into a user specified location.

Comparison of StarOS and Medusa results reveals that the accuracy of time measurements is operating system dependent. The wider spread of the StarOS clock reading results even when ignoring interrupts suggests that it is more difficult to get accurate time measurements from StarOS and that any StarOS experiments using Kmap operations probably have higher variability in execution times.

A lesson learned from this study of the clock measurements is that performance measurements must be done very carefully since even the most obvious items such as the clocks can fail to perform as expected.

4 Methodologies for measuring time

Since the inaccuracy of the clock routines for Cm* is mainly due to Kmap load and clock contention, corrective measures can compensate the incorrect clock readings by accounting for the Kmap load and clock contention during a measurement. Based on this premise, this section discusses methods that can be used to obtain more accurate time measurements for performance evaluation. More specifically, this section proposes a way to generate a repeatable workload for the system on which performance evaluation is done, develops methods for organizing performance evaluation experiments, presents algorithms to compute the net elapsed time given inaccurate clock readings, and tests these methodologies for validity.

4.1 Methodology of performance evaluation

In this project, workloads are synthesized by replicating the measured experiment. For example, to measure the performance of the message facility, the synthetic load will be the number of pairs of processes communicating with each other through messages.

The generation of a synthetic workload can be best illustrated by an example. Assume the execution time of a software routine, R, is to be measured under different system loads. The experiment then consists of a Cm executing R, while a number of other Cm's executing R constitute

the synthetic load. The result is the execution time of R as a function of the number of Cm's executing R simultaneously.

A basic approach for measuring performance is to have N identical experiments running in the system. The system workload is parametrized by the value of N and by how the experiments are distributed within the system. A simple way to measure performance is to have only one experiment per cluster that reads the clock. This means of all the experiments running in a cluster, only one experiment actually reads the clock to measure performance. This reduces the number of clock reads generated, produces less system load due to fewer clock reads, and results in the improved performance of both the clock reading software and the measured experiments. The decision to measure only one experiment per cluster is based on the assumption that all the Cm's have identical execution speed, and the symmetry of the Cm* architecture makes Cm's from the same cluster virtually indistinguishable from each other[4]. The validity of the assumption that all Cm's have the same execution speed will be shown in a later subsection.

Also, the timed experiment does not execute continuously. Rather, it is "injected" into the system at fixed intervals. This further reduces the amount of data generated. By injecting the timed experiment after the start up transients have decayed and the system workload has stabilized, more accurate results can be obtained. In the real situation, the user is often interested in finding out the execution time of a piece of software if he were to insert it in a system of a given workload. This situation is quite accurately modelled by the injection approach. Note that the effects of the transients caused by injecting an experiment have not been studied and may be a subject worthwhile for future research.

4.2 Methodologies for measuring elapsed time (Clock compensation)

Given methods to organize performance evaluation experiments, the next step is to develop algorithms for accurately measuring elapsed time. Early in this section, it was postulated that one can monitor the Kmap workload to improve the accuracy of elapsed time measurements. Below are two such algorithms.

---

[4] Actually; some Cm's are connected to I/O devices which may affect their performance. In all experiments, Cm's with I/O devices such as serial lines or Ethernet interfaces must not be used

Since reading the clock twice successively yields a result with a mean and variance that are both functions of the system load, the net elapsed time of any experiment can be computed by subtracting the average value of the elapsed time between two clock reads from the measured result. Using this algorithm, the expected value of the computed result equals to the true elapsed time, while the distribution of the computed result is identical to the distribution of the measured result. We shall refer to this algorithm as the *long term averaging technique.*

Since the load on the system is a time varying function, and since tasks performed by the system take time to complete, it is reasonable to assume that the system load at times separated by small intervals should be highly correlated. Because the time elapsed between two clock reads is a function of load, the autocorrelation of this elapsed time for short time intervals should also be high. Based on this assumption, the *short term averaging technique* approximates the time required to read the clock during an experiment by using the elapsed time between two successive clock reads that occur closely in time. Below is a mathematical analysis of the short term averaging technique.

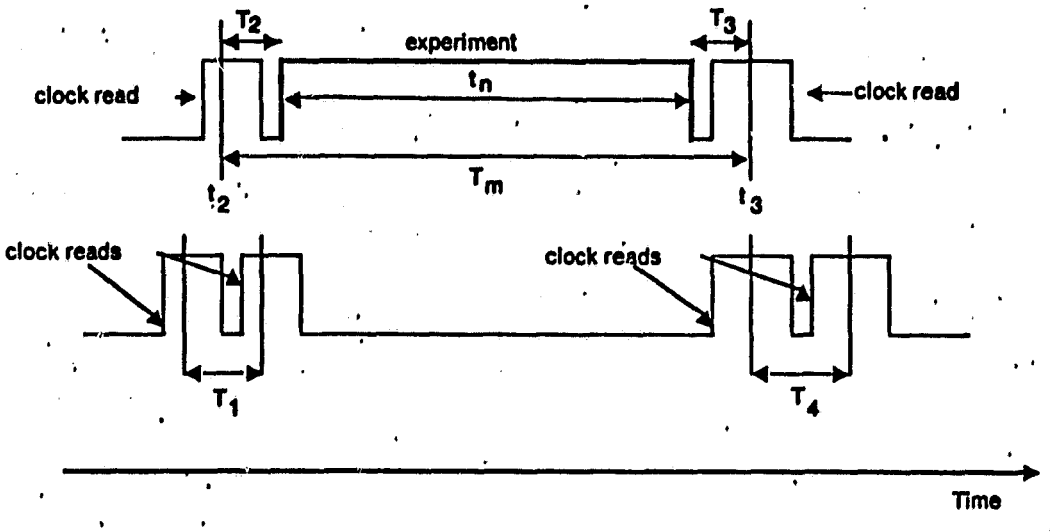

Figure 8: Short term averaging algorithm

Assume we are measuring the execution time of an experiment as illustrated in Figure 8. Then the variables are defined as follows[5]:

[5] Capital letters denote random variables

- t_n be the true elapsed time of the experiment.
- T_A is the computed elapsed time. (T_A approximates t_n.)
- T_m be the measured elapsed time.
- T_2 be the time interval between the moment the clock is read and the moment the experiment begins.
- T_3 be the time interval between the moment the experiment ends and the moment the clock is read again.
- T_1 and T_4 be the elapsed times between two pairs of clock reads.

Then $t_n = T_m - (T_2 + T_3)$ and

$$T_A = T_m - (T_1 + T_4)/2. \qquad (1)$$

But

$$T_m = t_n + T_2 + T_3,$$

therefore

$$T_A = t_n + T_2 + T_3 - (T_1 + T_4)/2.$$

If the expected values

$$E(T_1) = E(T_4) = E(T_2 + T_3) = \delta,$$

then

$$E(T_A) = t_n + \delta - (\delta + \delta)/2 = t_n.$$

Therefore, the expected value of the computed result equals the true result.

The variance of the computed result is:

$$V(T_A) = V(t_n + T_2 + T_3 - (T_1 + T_4)/2).$$

Let

$$V(T_1) = \sigma_1^2,\ V(T_4) = \sigma_4^2,$$

$$V(T_2) = \sigma_2^2, \text{ and } V(T_3) = \sigma_3^2,$$

then[6]

---

[6] If $X = \sum_{i=1}^{n} a_i Y_i$, then $V(X) = \sum_{i=1}^{n} a_i^2 V(Y_i) + 2\sum\sum_{i<j} a_i a_j \rho_{ij} \sigma_i \sigma_j$.

$$V(T_A) = \sigma_2^2 + \sigma_3^2 + \sigma_1^2/4 + \sigma_4^2/4 + 2\rho_{T_2,T_3}\sigma_2\sigma_3 - \rho_{T_1,T_2}\sigma_1\sigma_2 - \rho_{T_1,T_3}\sigma_1\sigma_3 - \rho_{T_2,T_4}\sigma_4\sigma_2 - \rho_{T_3,T_4}\sigma_4\sigma_3 + \rho_{T_1,T_4}\sigma_1\sigma_4 / 2. \quad (2)$$

where ρ_{T_i,T_j} is the correlation coefficient of the random variables T_i and T_j.

To simplify Equation (2), the following assumptions are made:

$$E(T_2) \simeq E(T_1) / 2, E(T_3) \simeq E(T_4) / 2, V(T_1) = V(T_4) = \sigma^2. \quad (3)$$

Now,

$$V(T_A) = \sigma^2[1 + 0.5\rho_{T_2,T_3} - 0.5\rho_{T_1,T_2} - 0.5\rho_{T_3,T_1} - 0.5\rho_{T_2,T_4} - 0.5\rho_{T_3,T_4} + 0.5\rho_{T_1,T_4}]. \quad (4)$$

If all correlation coefficients are unity, then

$$V(T_A) = 0.$$

In the worst case when $\rho_{T_1,T_2} = \rho_{T_3,T_1} = \rho_{T_2,T_4} = \rho_{T_3,T_4} = 0$, and $\rho_{T_2,T_3} = \rho_{T_1,T_4} = 1$,

$$V(T_A) = 2\sigma^2$$

for all non-negative correlation coefficients.

For very short interval measurements, the time-stamps t_2 and t_3 are very close together and the random variables T_2 and T_3 can be replaced by a new T_3 equals to the old $T_2 + T_3$. Then $\sigma_2^2 = 0$ and $\sigma_3^2 = \sigma^2$. We now have

$$V(T_A) = 1.5\sigma^2 - \rho_{T_1,T_3}\sigma^2 - \rho_{T_3,T_4}\sigma^2 + 0.5\rho_{T_1,T_4}\sigma^2. \quad (5)$$

If all correlation coefficients are unity, then

$$V(T_A) = 0.$$

If all correlation coefficients are zero, then

$$V(T_A) = 1.5\sigma^2.$$

In the worst case when $\rho_{T_1,T_3} = \rho_{T_3,T_4} = 0$, and $\rho_{T_1,T_4} = 1$,

$$V(T_A) = 2\sigma^2$$

for all non-negative correlation coefficients.

Even though Equation (2) expresses the value of the variance of the result, it cannot be solved unless the variances of T_2 and T_3 are known. In our case, this information is not available from the

experiment. To simplify the problem, the time elapsed between the issuing of a clock read to the actual reading of the clock and the time elapsed between the reading of the clock to the returning of the result to the reader are assumed to be have the same mean and variance. Equation (4) then expresses the variance of the result. Equation (5) applies when the duration of the experiment to be measured is extremely short and close to zero.

This algorithm shows that for any method used to select the two pairs of clock reads, the worst case will yield a result with a variance twice the variance of the elapsed time between two clock reads. In the best case, the variance of the result is zero. In cases where assumptions of Equation (3) apply, zero variance in the result is obtained when the correlation coefficients between T_1 and T_2 (ρ_{T_1,T_2}) and between T_3 and T_4 (ρ_{T_3,T_4}) are both unity.

A way to evaluate the methods used to select the clock read pairs is to compute the improvement factor k. In any experiment that measures a fixed time interval, let $V(T_A)$ be the variance of the corrected result and let σ^2 be the variance of the uncorrected result. Then k is defined such that

$$k = \sigma^2 / V(T_A).$$

The larger the value of k is, the better the improvement. The range of k is between 0.5 and infinity. When k is unity, the variance of the corrected result is unchanged. Note that the Long Term Averaging algorithm always yields a k of unity.

The objective for selecting the two pairs of clock reads for compensation is to maximize the correlation coefficients ρ_{T_1,T_2} and ρ_{T_3,T_4}. Though there are many ways this can be done, only two methods will be presented. The reader is encouraged to design his own implementation, bearing in mind that the objective is to maximize the correlation coefficients stated above.

The first method presented, hereafter referred as Method I, is illustrated in Figure 9. If the processor that starts the measurement reads the clock twice at the beginning of the experiment and the processor that terminates the measurement reads the clock twice after the experiment, then T_1 should highly correlate with T_2 while T_3 should highly correlate with T_4. This is because these clock reads occur very closely in time.

This method has the advantage that no clock read occurs during the experiment and therefore the performance of the experiment under measurement is not affected.

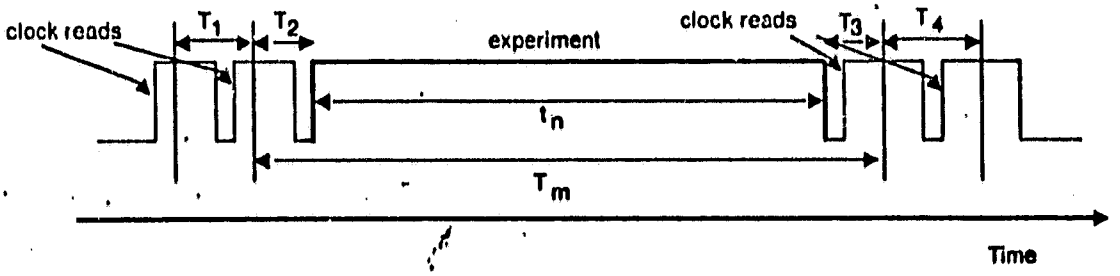

Figure 9: Short term averaging, Method I

The second method presented is referred as Method II, as illustrated in Figure 10. If a clock process runs concurrently with the experiment and periodically samples the load by reading the clock twice in succession, the elapsed time can be used for compensation. One approach is to select the clock read pair of the clock process that is closest in time to the clock read that starts the measurement to give T_1, and to select the clock read pair of the clock process that is closest in time to the clock read that stops the measurement to give T_4.

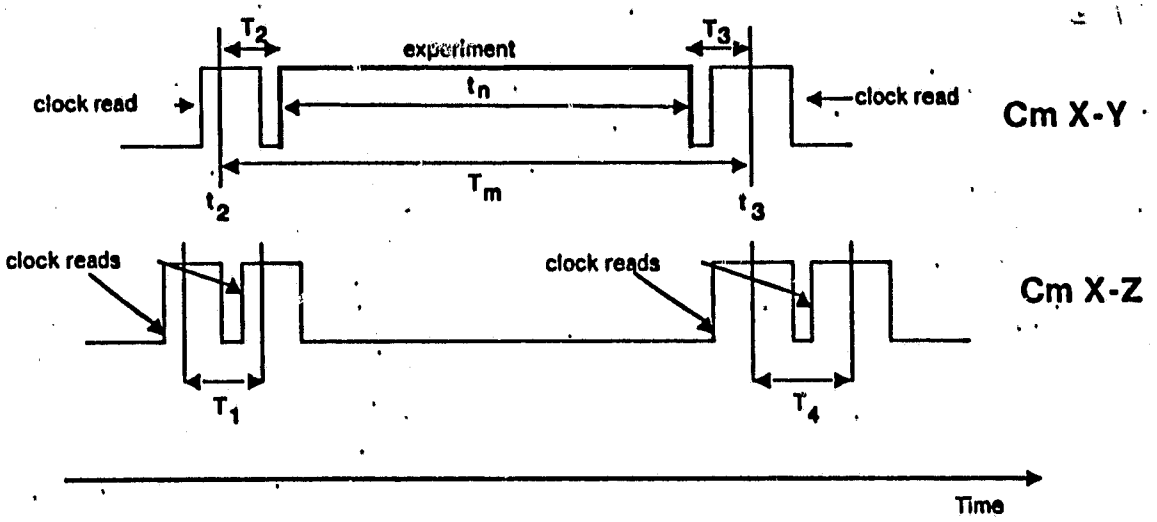

Figure 10: Short term algorithm, Method II

This method is less desirable than the previous method because of its added complexity and because of its interference with the performance of the experiment by the presence of a clock

reading process. However, subsequent sections will show that this method yields quite accurate results.

4.3 Execution speed of computer modules

The method for generating workload assumes that all the computer modules in Cm* execute at the same speed. To verify this assumption, an experiment was set up to measure the execution speed of the computer modules in Cm*. This experiment involved timing the execution of a piece of code stored in the local memory of each Cm. Once the program execution begins, the Kmaps are not involved.

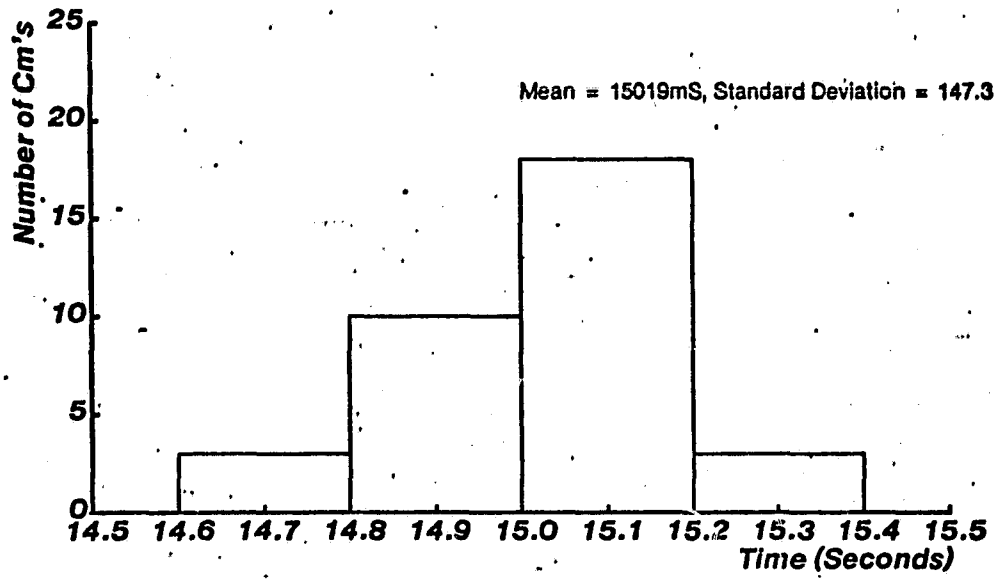

Figure 11: Histogram of execution time of 34 Cm's

The results show that all of the thirty-four computer modules tested had execution speeds within 4.6% of each other. A histogram of the execution speed of the computer modules is shown in Figure 11. The conclusion for this experiment is that every computer module can be considered to have essentially identical execution speed, therefore experiments performed on any computer module should be equally valid.

4.4 Evaluation of clock reading compensation techniques (Method I)

The methods to compensate the clock readings cannot be rigorously proved to produce correct result because they employ only heuristic approaches. Therefore, to validate our methods, an attempt is made only to show that an accurate result for a fixed interval measurement (e.g., 0 seconds) is obtained under some reasonable system load.

The experiment to validate Method I consists of a process reading the clock four times in succession. The first two clock reads are used to compute T_1, the second and third clock reads measure a null experiment which has zero execution time. The third and fourth clock reads are used to compute T_4. The synthetic workload is generated by replicating a large number of processes distributed evenly among the clusters reading the system clock. The experiment was performed for both StarOS and Medusa.

Figure 12 illustrates the result of the experiment using the StarOS 4-Read clock routine to measure time. The solid curve is the distribution density of the compensated result, while the dashed curve is the distribution density of the result before correction is applied. The ideal result is an impulse of unit magnitude at 0μS. The mean compensated result was -1.90μS, and the improvement factor, k, was 0.8. Recall that for $k < 1$, the variance of the result is increased. The same experiment using the 1-Read clock routine gave an improvement factor of 0.81.

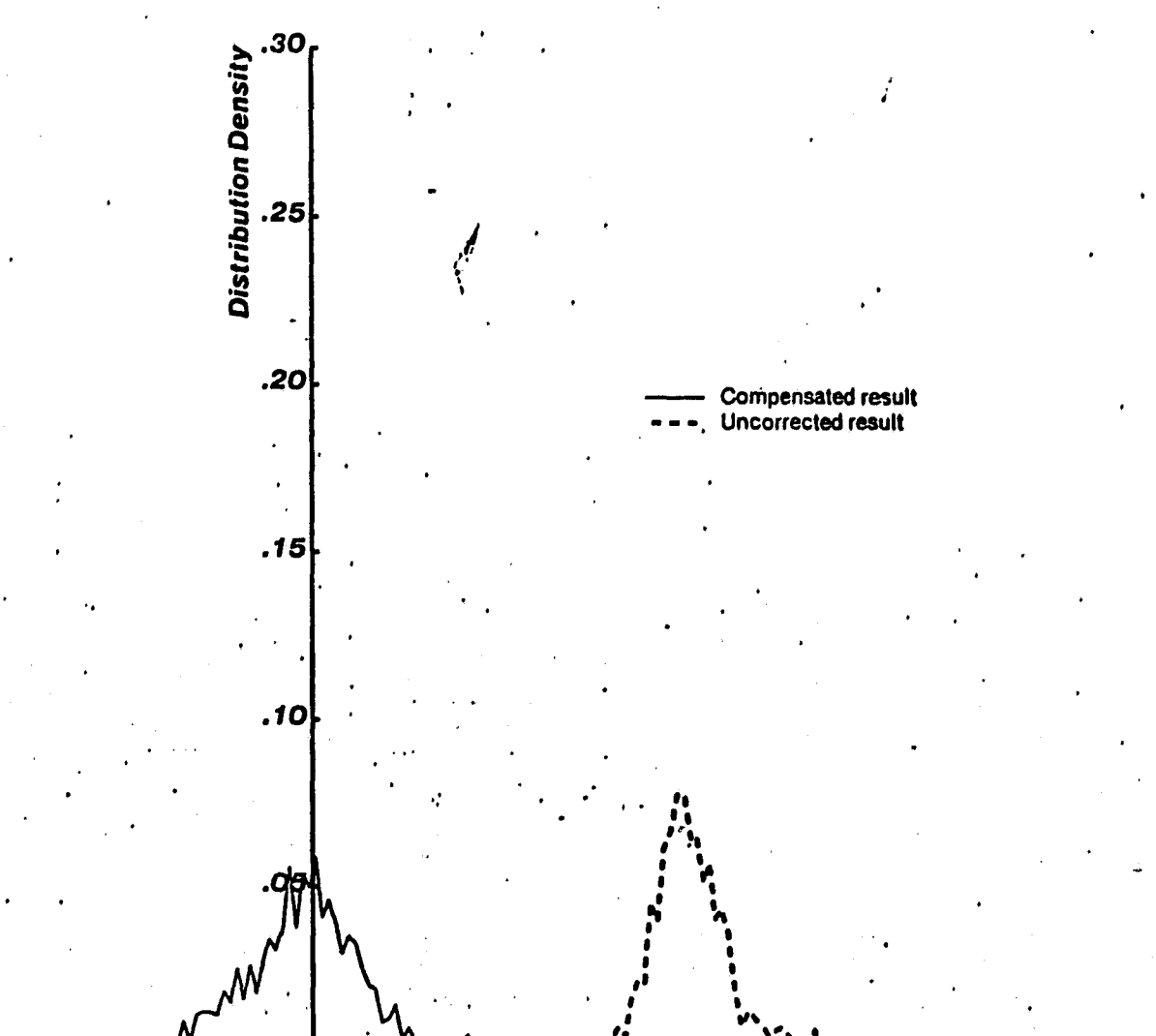

Figure 12: Measuring zero elapsed time using Method I with 4-Read routine

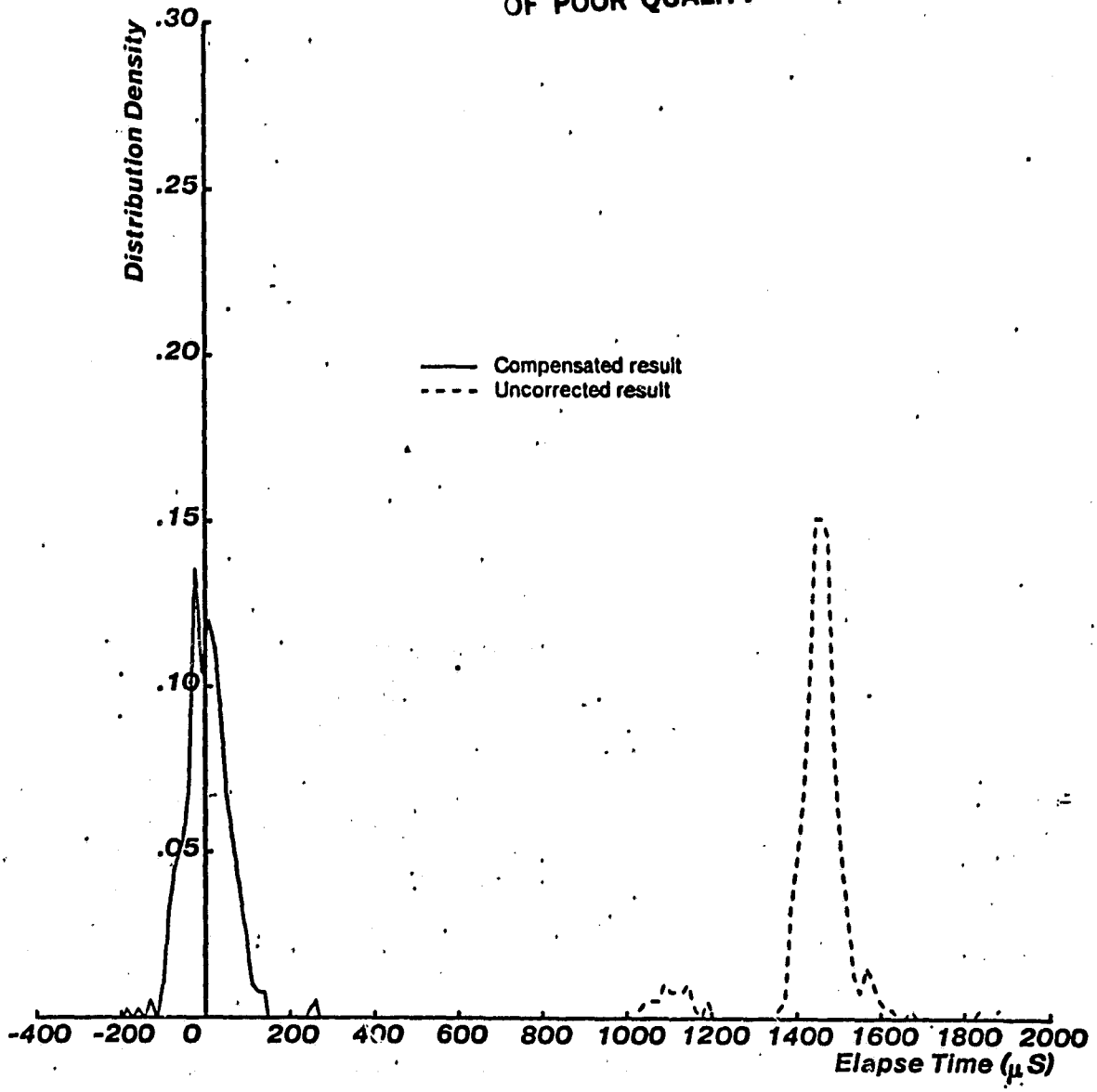

Figure 13: Measuring zero elapsed time using Method I with Medusa 4-Read routine

Figure 13 illustrates the result under Medusa using the 4-Read clock routine to measure time. The mean compensated result was 6.69μS, and the improvement factor k was 3.57. This represents a great improvement in the variance of the results. The 1-Read clock routine gave an improvement factor of 0.68.

4.5 Evaluation of clock reading compensation techniques (Method II)

The experiment that validates Method II consists of a clock process executing in a computer module from the cluster where the experiment is performed, a process that does two successive clock reads to measure the elapsed time (which should ideally be zero if reading the clock does not take any time), and a number of pairs of communicating processes that send each other messages to create a synthetic system workload. Each pair of these communicating processes is independent of the other processes in the system, and their sole purpose is to generate load to the Kmaps through which clock read requests are routed. The experiment process measuring zero elapsed time is synchronized with the clock process. It signals the clock process to start reading the clock, reads the clock twice successively, and then sends the results of the two clock reads to the clock process which computes the net elapsed time according to Equation (1).

Figure 14 shows the distribution density of the results of the StarOS experiment. The dashed curve is the result of the measured reading (T_m in Equation (1) and Figure 10). The solid curve is the result after Method II has been applied (T_A in Equation (1)). The results were taken from 1000 repetitions of the experiment. The mean value was -5.24μS, while the improvement factor k was 1.14. The improvement factor for the 1-Read clock routine was 1.98.

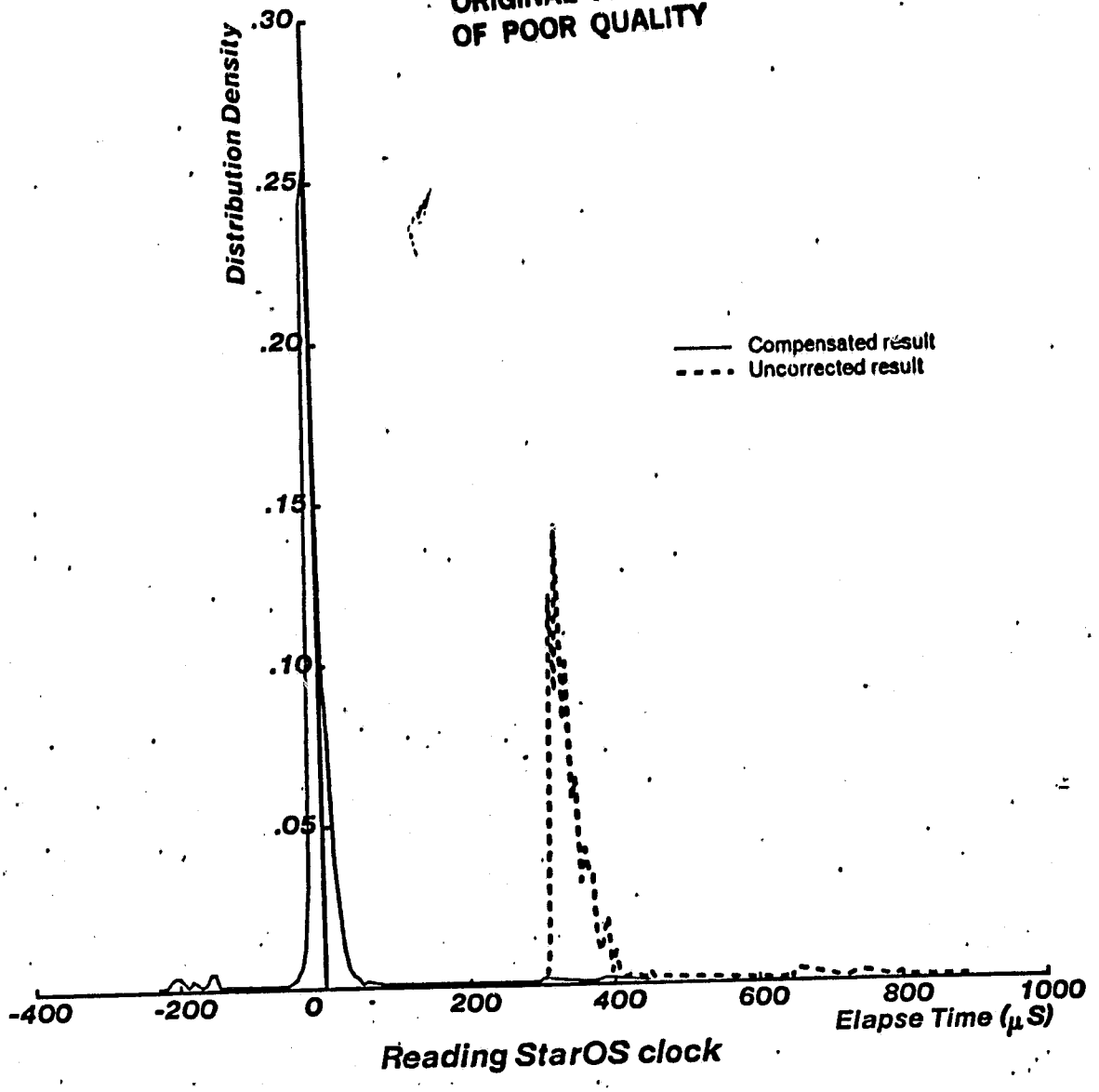

Figure 14: Measuring zero elapsed time using Method II with StarOS 4-Read routine

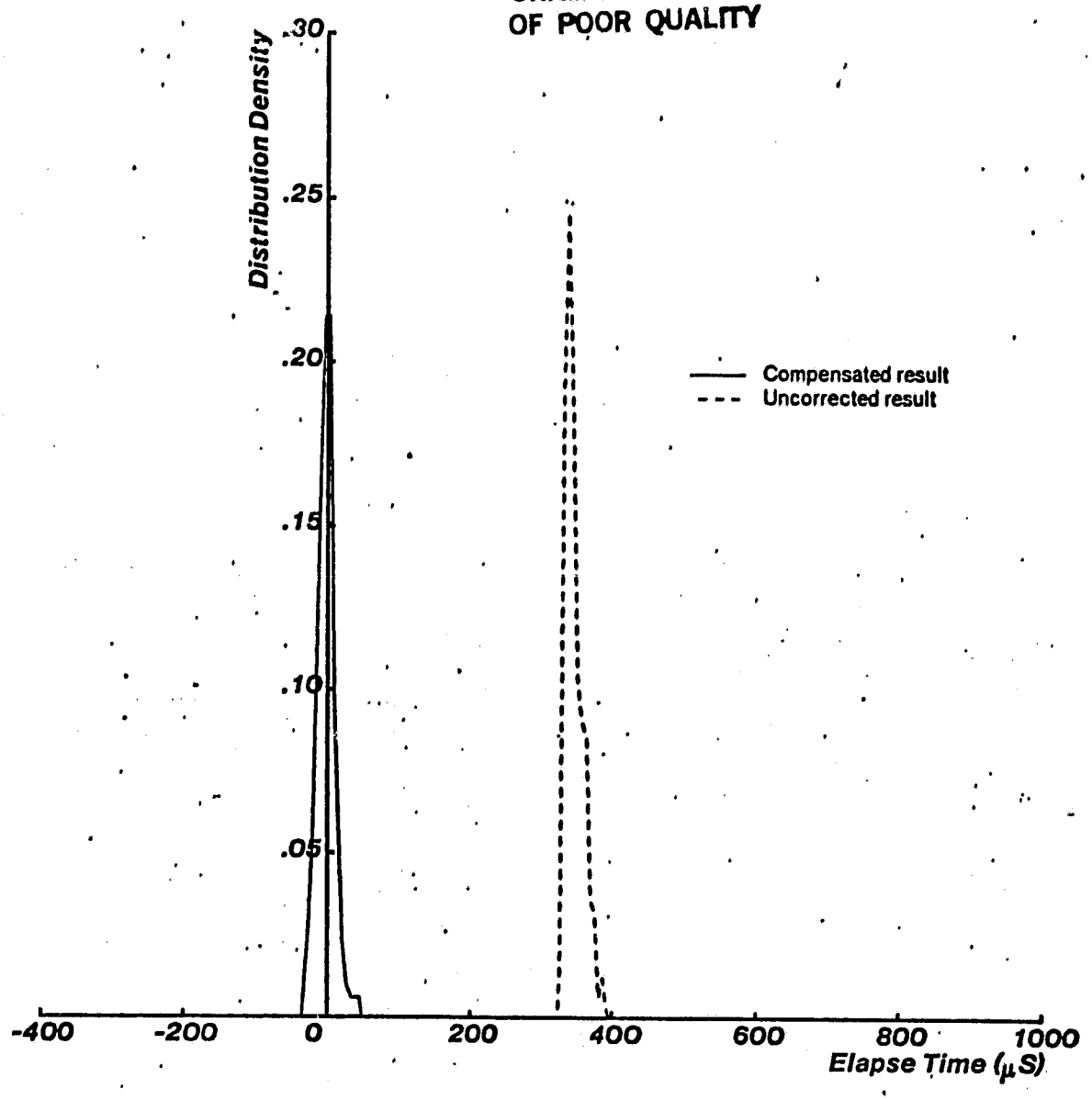

Figure 15: Measuring zero elapsed time using Method II with Medusa 4-Read routine

When executing under Medusa, the experiment yields different results. Figure 15 shows the distribution density of the Medusa experiment. The mean value of the compensated result was 6.69μS and the improvement factor was 1.11. The improvement factor was 0.81 for the 1-Read clock routine.

4.6 Discussion of results

As shown by the result of the above eight experiments, the mean corrected value was less than 6.7μS. This provides an upper bound to the accuracy of measurement obtainable. It is concluded that these measurement methods are not suitable for measuring elapsed times that are less than fifty microseconds because the relative error for small interval measurements is high.

Of the eight experiments performed, four showed improvement in the variance of the corrected result (with the improvement factor k ranging from 1.11 to 3.57). The other four cases showed a k less than 1 but greater than 0.67. Recall in Equation (4), it was shown that the worst case k would be 0.5, while if the clock reads used for compensation were totally uncorrelated to the clock reads that they were supposed to compensate, the value of k would be 0.67. In the Medusa experiment using Method I with the 1-Read clock routine, the value of k was 0.68. This shows that during that experiment, the system load was changing so rapidly that the execution time of any clock read was essentially uncorrelated to the execution time of any previous clock reads or subsequent clock reads.

It is interesting to note that three out of the four experiments using Method II resulted in improved variance, while only one out of the four experiments using Method I resulted in the improved variance. This phenomenon is mostly due to the difference in the type of system load. In all the experiments using Method II, the system workload was the load created by a large number of processes sending and receiving messages. Since sending and receiving of messages are lengthy processes (on the order of a millisecond), the load of the system is trackable by the clock reads. When the granularity of the system load decreases to a duration comparable or shorter than the time required to execute a clock read, the tracking of the system load using clock reads fails. This was the case for the experiments to validate Method I. The synthetic workload was a large number of processes reading the clock. Because the load on the system had the same duration as the clock reads used to sample the load on the system, the tracking of the system load failed.

A problem with Method I is that it does not track system load correctly when an interrupt occurs. This is because an interrupt during an experiment will either affect the clock read used to obtain the measurement or the clock read used to obtain the compensator, but not both. This explains why Method I did not work very well under StarOS since the StarOS processes were interrupted sixty times per second. Method II tracks well even with interrupts. This is because

interrupts by the line time clocks are system-wide, therefore an interrupt affecting the clock read used to obtain the measurement is likely to occur in the clock process as well. This means that the extra time required to handle an interrupt during a clock read is likely to be compensated for.

4.7 Conclusion

In this section, methods have been developed to measure the performance of Cm* software under system load. Two algorithms have been developed to yield more accurate elapsed time measurements than the clock routines can provide.

Experiments were performed to validate the measurement methodologies. The variation of execution speed among different Cm's was found to be around 4.6%. The long term algorithm developed to compensate for clock readings has a very predictable behavior and no experiment was performed to test its validity. The short term algorithm was implemented with two variations - Method I and Method II. Experiments were set up to evaluate both methods. The base line accuracy of these methods was around 6.7μS. Therefore these methods are not suitable for measuring short duration events (50μS or less), but are perfectly suitable for measuring longer duration events such as operating system calls.

It was noted in Section 3 that the 2μS resolution of the clock was not usable under heavy system loads because of the uncertainty in communication delay, and that higher values of clock tick could be used. In this section, it is shown that the accuracy of the clock reading results are so improved that the 2μS resolution is usable.

Because Method I was theoretically superior to Method II, it was given the tough task of executing under system load of very small granularity. Results showed that Method I was unable to perform properly in small granular system loads. Method II was tested with a more reasonable load and was found to perform quite well. The short term algorithm using Method II performed better than the long term algorithm would have performed in three of the four experiments tested. Because Method I is more desirable than Method II in that it does not affect the experiment under measurement, it is believed that Method I should perform at least as well as Method II under a reasonable system load provided that there are no interrupts. When there are interrupts, Method II is the preferred method.

An important conclusion is that it is not possible to present a clock compensation scheme that works under arbitrary system load because the clock readings can only sample the system load at a finite rate. The reader is encouraged to develop his own clock compensation technique. However, he should test his scheme to ensure that it tracks the system load reasonably well. The zero elapsed time measuring experiment is recommended for such testing. Below is the procedure for testing a clock compensation method.

```
DO
     Pick a clock compensation method;
     DO
         Try it out in experiment and measure zero elapsed time;
         IF results not satisfactory THEN
             Fine tune the method;
     UNTIL method is optimal or results satisfactory;
UNTIL exhausted all methods or results satisfactory;
```

5 An example experiment

This experiment evaluates two performance measures of a message-based operating system. The first measure is the latency of the message mechanism. Latency is defined as the time elapsed from the moment a sender begins to send a message to the moment the receiver receives the message. The second measure is the execution time of a message-based remote procedure call (RPC).

While a message mechanism is often provided by an operating system as a primitive for interprocess communication (IPC), remote procedure calls are often provided at the language level [7] [14]. The two are related in that RPC's are often built on the message mechanism. The remote procedure call of this experiment consists of a *client* who sends a message containing the arguments for the call, and a *server* who receives the message, performs the specified function, and returns a message containing the result. The time elapsed from the moment the client begins to send a message to the moment it finishes accessing the returned result constitutes the execution time of the RPC. While this simple RPC does not perform any type checking, error recovery, etc., it is a simplified model of the RPC's of real systems and can help indicate the RPC performance of a system.

5.1 Organization of experiment

The experiment consists of N client/server pairs for N is greater than or equal to one. Below is the pseudo-code of such a client/server pair:

```
client:
        prepare argument
        T1                                    server:
        send the arguments ------------------> wait for a message
                                               T2
                                               access parameters
                                               perform computation
        wait for the results <---------------- send results
        access results
        T3
```

The latency of a message is T2 - T1, while the execution time of the RPC is T3 - T1. For the latency measurement to be meaningful, the server must be blocked before the client sends a message.

The experiment is implemented under StarOS and consists of a master process which spawns client/server pairs in locations specified by the user. Since both processes of a client/server pair reside in the same cluster, all communications within a client/server pair are intracluster. Of all the client/server pairs spawned, only one pair reads the clock to measure performance. This pair is responsible for sending all its results to the master process. The master process ships the results to a VAX/UNIX system via the Ethernet for storage and analysis. The clock compensation technique used is Method II of the short term averaging algorithm as presented in Section 4.

5.2 Experiments

The experiment was performed with different levels of load ranging from one client/server pair per cluster to three client/server pairs per clusters. The total number of words accessed varied from 0 to 200 in increments of 50. All measurements were repeated 512 times.

The measurement of zero elapsed time was performed during each repetition of the experiment as a run-time check to see how well the clock compensation scheme was tracking the system load. It was found that the mean error was no worse than 4.9μS, while the improvement factor was between 0.74 and 0.78. This low improvement factor was due to interrupts that were not trackable by

Method I. By simulating the situation that there are no interrupts, the improvement factor was between 1.01 and 1.22.

5.3 Results

5.3.1 Latency measurements

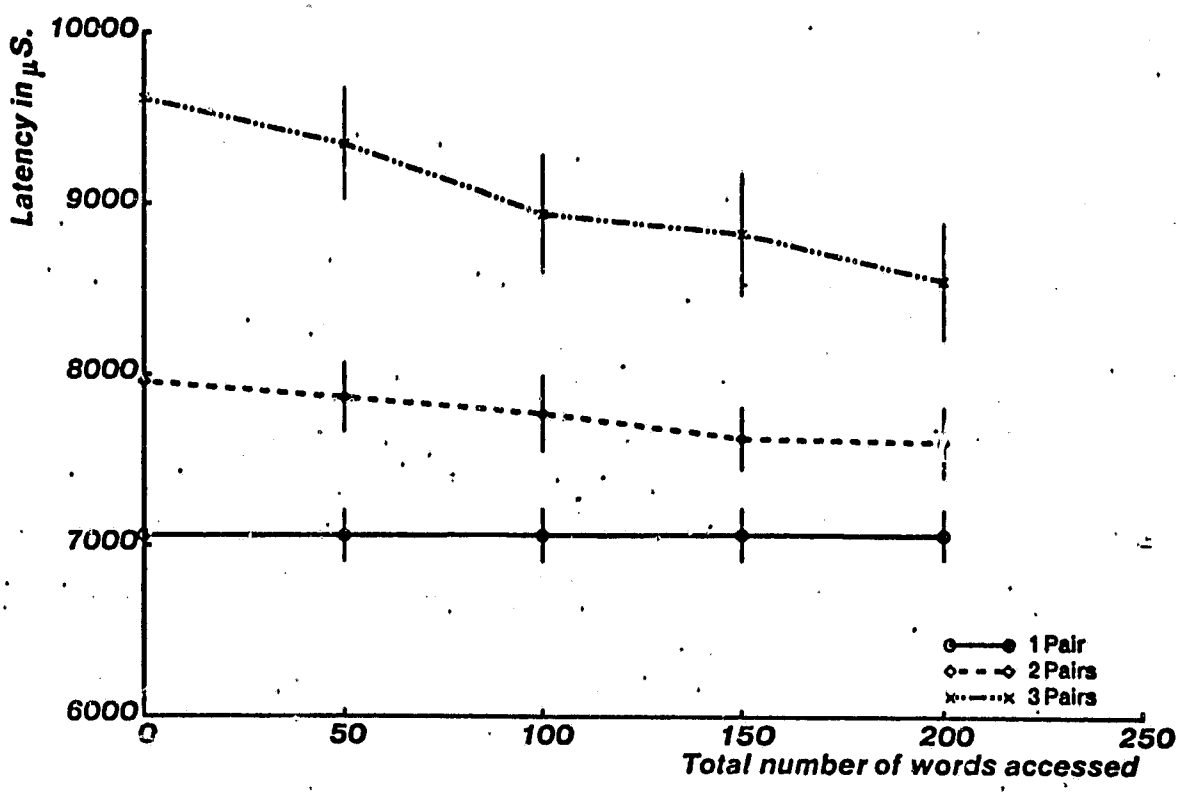

Figure 16: Latency of StarOS messages in the experiment

Figure 16 illustrates the latency of the StarOS messages in our experiment. The solid curve shows that latency is constant at approximately 7050μS and is independent of the total number of message words accessed. The dashed curve shows that when two client/server pairs are executing in the same cluster, the latency rises to 7960μS because of increased Kmap load. As the total number of message words accessed increases, more time is spent accessing remote memory, resulting in fewer RPC executions per unit time. This causes a decrease in the rate of message operations, resulting in a decreased latency. The broken curve shows that latency is 9615μS when

three client/server pairs are executing in the same cluster. The vertical bars at each point of the curves show the magnitude of the standard deviation at that point.

5.3.2 Execution time of RPC

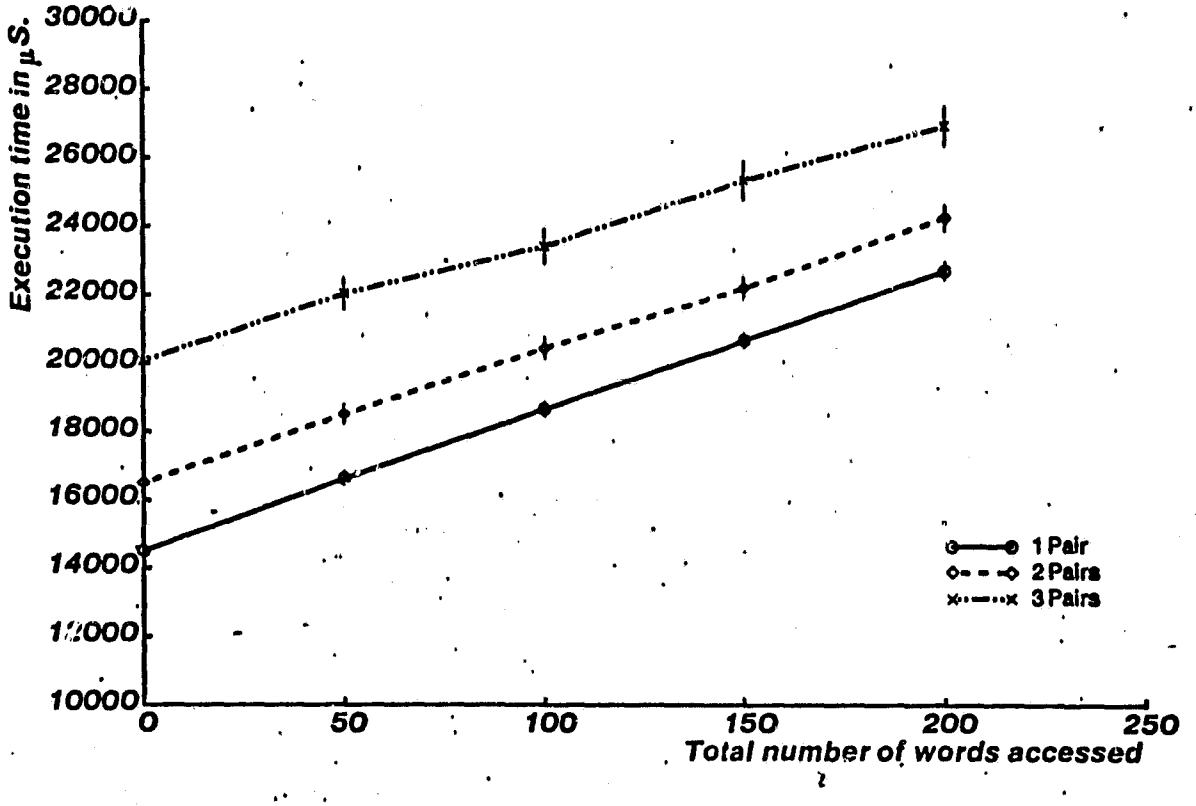

Figure 17: RPC execution time versus the total number of words accessed

Figure 17 shows the execution time of the remote procedure call as a function of the total number of words accessed by the client and the server. The execution time of the call is a linear function of the number of words accessed. The solid curve shows the mean execution time of the RPC with the number of message words accessed ranging from 0 to 200. The slope of the solid

curve is 49.9μS per word[7]. The dashed curve shows the result of the same experiment with two client/server pairs executing in a cluster. The broken curve shows the results for three client/server pairs.

5.4 Conclusion

In this section, an example of measuring StarOS message latency and message-based RPC execution time was presented. This example experiment implemented two ideas developed in Section 4. First, it generated system workload by replicating the experiment in different parts of the system. The number of replicas and how they were distributed in the system were both controlled by the experimenter at program run time. Second, the experiment employed one of the clock compensation techniques developed in Section 4. The addition of the clock compensation technique into the basic experiment required only the addition of a subroutine which computed net elapsed times given four time-stamps.

6 Conclusion

This project discovered the erratic behavior of the Cm* clock reading software and presented an alternate set of clock reading software. Additionally, it recommended that the length of a clock tick should be set to be commensurate with the variation in communication delays. It proposed that a Kmap operation be provided to latch the clock value and to read the clock register indivisibly. Most important of all, it provided a clock compensation scheme for measuring elapsed time with an accuracy much greater than that provided by the clock reading software.

However, in developing the mathematical model of the clock compensation techniques, it was assumed that the correlation between the time elapsed for two successive clock reads and the system load was non-negative, and that the autocorrelation of system loads separated by short time intervals was non-negative. The validity of these assumptions should be investigated. Also, the

---

[7]This value must not be interpreted as the intracluster memory access time for StarOS. Rather, it is the time required for an iteration through the following Bliss-11 program loop:

```
.INCR k FROM 0 TO .lreused- 1 DO
        temp = .ResultPage[.k];
```

This loop compiles into six LSI-11 instructions, with one of them performing a remote memory reference.

clock compensation technique fails when the granularity of system load is too small. At present, little is known about the granularity and the time profile of system load. Further study is required to gain this knowledge.

Of more general concern, this project stressed the importance of real-time clock designs in multiprocessors since they greatly affect the clock's usability. Future multiprocessor designs should include a globally readable clock that is accessed through a special bus unaffected by system load and contention. An example of such a clock is the system clock of C.mmp multiprocessor. When such a design is not feasible, the clock should be of the same width as the data bus so that the entire clock word can be read in one access. If this is not practicable, there should be an instruction that latches the clock value and reads it indivisibly. The implementation of such an instruction should be straight forward when the clock register size is compatible with one of the machine supported data types, since multiprocessor should allow indivisible read/write accesses to all machine supported data types to guarantee data consistency. Thus the instruction in essence is simply a latch operation followed by a read operation.

For multiprocessor systems using a single global clock and experiencing the same problems experienced by Cm*, this project provided a general scheme for measuring elapsed time accurately. The study of Cm* clock performance shows that system load can be gauged by reading the system clock. This is because reading the clock exercises many of the system resources (Map-bus, intercluster bus, Kmaps, etc.). This idea of exercising system resources to measure their load is worth exploring.

References

[1] C. Ellingson & R. J. Kulpinski.
Dissemination of System Time.
IEEE Trans. Comm. Com. 23(5):605-624, May, 1973.

[2] E. F. Gehringer and R. J. Chansler, Jr.
StarOS User and System Structure Manual.
Technical Report, Carnegie-Mellon University, Computer Science Department, June, 1981.

[3] A. K. Jones and E. F. Gehringer, editors.
The Cm Multiprocessor Project: A Research Review.*
Technical Report, Carnegie-Mellon University, Computer Science Department, July, 1980.

[4] Thomas H. Kong.
Measuring Time for Performance Evaluation of Multiprocessor Systems.
Master's thesis, Carnegie-Mellon University Department of Electrical Engineering, November, 1982.

[5] Leslie Lamport.
Time, Clocks, and the Ordering of Events in a Distributed System.
Communications of the ACM 21(7):558-565, July, 1978.

[6] Madhav V. Marathe.
Performance Evaluation at the Hardware Architecture Level and the Operating System Kernel Design Level.
PhD thesis, Carnegie-Mellon University Computer Science Department, December, 1977.

[7] Bruce Jay Nelson.
Remote Procedure Call.
PhD thesis, Carnegie-Mellon University Computer Science Department, May, 1981.

[8] J. K. Ousterhout, D. A. Scelza and P. S. Sindhu.
Medusa: An Experiment in Distributed Operating System Structure.
Communications of the ACM 23(2), February, 1980.

[9] John K. Ousterhout.
Partitioning and Cooperation in a Distributed Multiprocessor Operating System: Medusa.
PhD thesis, Carnegie-Mellon University, Computer Science Department, April, 1980.

[10] Levy Raskin.
Performance Evaluation of Multiple Processor Systems.
PhD thesis, Carnegie-Mellon University Department of Electrical Engineering, August, 1978.

[11] D. M. Ritchie and K. Thompson.
The UNIX Time-Sharing System.
Communications of the ACM 17(7):365-375, July, 1974.

[12] Richard Snodgrass.
SIMON - A Simple Monitor for StarOS.
Carnegie-Mellon University Computer Science Department, Internal report, 16 February 1981.

[13] Richard Snodgrass.
Monitoring Distributed Systems - Thesis Proposal.
Carnegie-Mellon University Computer Science Department, 10 November 1980.

[14] Alfred Z. Spector.
Performing Remote Operations Efficiently on a Local Computer Network.
Communications of the ACM 25(4), April, 1982.

[15] William A. Wulf, Roy Levin, and Samuel P. Harbison.
Hydra/C.mmp: An Experimental Computer System.
McGraw-Hill, 1981.

Software Voting in NMR Computer Structures

Gary York

Department of Electrical Engineering
Carnegie-Mellon University

17 December 1982

Table of Contents

List of Figures